Facultad de Informática de Barcelona

Lowering of OpenMP directives to the Intel® OpenMP* Runtime API

Autor: Raúl Peñacoba

Director: Eduard Ayguadé *Co-director:* Sergi Mateo

23 de enero de 2018

Abstract

After a brief overview of Open Multi-Processing programming model, this document aims to show the process of extending the current Mercurium's compiler infrastructure in order to lower tasking directives to the Intel OpenMP Runtime library.

In the first place, task directive lowering is explained. Then, other directives of syncronization, loop iteration division in tasks, and so on are showed.

Resum

Després d'una breu introducció a *Open Multi-Processing*, aquest document mostra el procés d'extendre la infraestructura del compilador *Mercurium* per suportar les directives de *tasking* a través del la libreria de Intel.

En primer lloc, s'explica el *lowering* de la directiva *task*. Després, es mostren altres directives de sincronització, divisió d'iteracions d'un bucle, etc.

Resumen

Después de una breve introducción a *Open Multi-Processing*, este documento muestra el proceso de extender la infraestructura del compilador *Mercurium* para soportar las directivas de *tasking* a través de la librería de Intel.

En primer lugar, se explica el *lowering* de la directiva *task*. Después, se muestran otras directivas de sincronización, división de iteraciones de un bucle, etc.

Agradecimientos

En primer lugar, me gustaría agradecer a mi director del proyecto Eduard Ayguadé, por darme la oportunidad de realizar este proyecto como trabajo de final de grado.

También, agradecer a Sergi Mateo, por enseñarme todo lo necesario para realizar este proyecto, resolver mis dudas y aconsejarme en los momentos que lo he necesitado. Sin su ayuda me habría sido muy difícil realizar el trabajo con éxito.

Por último, agradecer a mi familia y amigos por animarme y apoyarme en todo momento. Sin ellos no estaría aquí.

Índice

Índice de figuras

Índice de Tablas

Acronyms

API Aplication Programming Interface. 21, 24, 28, 30

AST Abstract Syntax Tree. 43, 46

BOE Boletin Oficial del Estado. 35

BSC - CNS Barcelona Supercomputing Center - Centro Nacional de Supercomputación. 20, 33, 37, 41

ECTS European Credit Transfer and Accumulation System. 29

FPGA Field Programmable Gate Arrays. 10

GCC GNU Compiler Collection. 20, 21, 33

GDB GNU Project Debugger. 34

GEP Gestió de Projectes. 29

HPC High Performance Computing. 10, 35

LLVM Low Level Virtual Machine. 20, 24

MPI Message Passing Interface. 10

OpenMP Open Multi-Processing. 11, 18, 20, 24, 26, 27, 30, 31, 40, 42

PAP Programación y Arquitecturas Paralelas. 20

PAR Paralelismo. 20

SSH Secure SHell. 31

VPN Virtual Private Network. 31

Glosario

Clang C *frontend* de LLVM. 21, 27, 33

embedded systems Sistemas de computación diseñados para realizar una o algunas pocas funciones dedicadas en tiempo real. 10

Git *software* de control de versiones, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones. 27, 31, 34

Mercurium Compilador *source to source* experimental que es utilizado junto al *runtime* Nanos, el cual soporta OmpSs, una extensión de OpenMP desarrollada por el BSC. 20, 24, 26, 30, 31, 33, 40, 42, 46, 80, 81

1. Introducción

Desde los inicios de la computación, la necesidad de procesar cada vez más información ha causado un aumento de la capacidad de cálculo de todos los dispositivos que nos rodean: *embedded systems* (como *smartphones*), ordenadores personales, consolas, *supercomputers*, etc.

En la actualidad, el uso de multiprocesadores es cada vez más común, así como el *software* que pretende sacarle partido. En el terreno de *High Performance Computing (HPC)*[1], además, es habitual ver estos dispositivos trabajando unos junto a otros, utilizando coprocesadores (tarjetas gráficas, etc), así como *Field Programmable Gate Arrays (FPGA)*[2] para mejorar todavía más el rendimiento.

La desventaja de todo esto es que, desarrollar *software* que trabaje con estos sistemas heterogéneos no suele ser tarea fácil. Conceptos como análisis de dependencias, particionado de datos y gestión de las comunicaciones son imprescindibles para aprovechar al máximo su potencial. La creación de modelos de programación abstraen ciertos aspectos dependientes de la arquitectura, facilitando el trabajo al desarrollador.

Dada la necesidad de compartir información en sistemas de memoria distribuida, *Message Passing Interface (MPI)*[3] surge para definir un protocolo que permita enviar y recibir información entre los dispositivos. El paradigma de paso de mensajes al que pertenece *MPI* ofrece una visión lógica formada por procesos, cada uno con su *address space*.

Como los datos pertenecientes a un espacio de direcciones no son visibles para el resto, la comunicación del envío y recepción de datos debe ser explícita. Esto permite que una vez los datos se hayan repartido entre los procesos, éstos puedan trabajar de forma independiente.

Por otro lado, programar siendo consciente de las comunicaciones que se deben realizar añade más carga de trabajo al desarrollador, aunque permite diseñar algoritmos muy eficientes en la mayoría de casos.

Otro pilar del *HPC* es el uso del modelo de programación de memoria compartida (o *shared memory*). *Open Multi-Processing (OpenMP)* pertenece a este paradigma. La ventaja que ofrece *OpenMP* respecto a *MPI* es que la comunicación que se realiza entre los procesadores es implícita y transparente. Esta abstracción permite que el sistema se pueda expandir añadiendo más procesadores sin tener que realizar modificaciones a nivel de código. Otra ventaja de *OpenMP* es que los cambios necesarios para paralelizar un código secuencial y obtener un buen rendimiento son mínimos. Únicamente deben añadirse un par de directivas al código.

Dado que *MPI* y *OpenMP* intentan resolver problemas diferentes (comunicación *internode* e *intranode*, respectivamente), ambos se usan conjuntamente para sacar el máximo partido al hardware disponible.

1.1. *OpenMP*

Como este proyecto está fuertemente relacionado con *OpenMP*, es imprescindible revisar algunos conceptos de la especificación de este modelo de programación. Para ello, los siguientes apartados introducirán algunas de las directivas de éste, haciendo énfasis en las directivas de *tasking*.

1.1.1. *parallel*

```
int main(void) {
    #pragma omp parallel num_threads(4)
    { printf("Thread %d: Hello World!\n", omp_get_thread_num()); }
}
```

```
Thread 0: Hello World!
Thread 3: Hello World!
Thread 2: Hello World!
Thread 1: Hello World!
```

Figura 1: Hola Mundo paralelo

Un programa *OpenMP* se ejecuta secuencialmente hasta la directiva *parallel*. Esta directiva es la encargada de crear un grupo de *threads*, cuya cantidad se puede especificar explícitamente en la directiva usando la cláusula *num_threads*, utilizando una variable de entorno, o mediante una llamada a función. El *thread* que se encuentra con la directiva se convierte en el *thread master* con identificador 0.

El resultado de este ejemplo es el mensaje `Thread + id: Hello World` escrito por los cuatro threads de la región paralela.

1.1.2. *single* y *barrier*

A menudo, dentro de una región paralela buscamos que cierto trozo de código se ejecute sólo por un *thread*. Un ejemplo sencillo de este caso es la suma de una lista de números. Cada *thread* puede realizar una suma parcial, y al final un *thread* hace la suma de las sumas parciales. Este último paso se puede efectuar con la directiva *single*.

Por otro lado, dentro de una región paralela se necesita habitualmente alguna forma de sincronización entre *threads*. La directiva *barrier* nos permite establecer un punto en el que los *threads* esperarán hasta que hayan llegado todos.

```
int main(void) {
    #pragma omp parallel num_threads(4)
    {
        printf("Thread %d: Hello World!\n", omp_get_thread_num());
        #pragma omp barrier
        #pragma omp single
        printf("Thread %d: I have the honor of printing
                this last message. Goodbye!\n", omp_get_thread_num());
    }
}
```

```
Thread 0: Hello World!
Thread 3: Hello World!
Thread 2: Hello World!
Thread 1: Hello World!
Thread 2: I have the honor of printing this last message. Goodbye!
```

Figura 2: Ejemplo *single* y *barrier*

En el ejemplo de la figura anterior, los *threads* se esperan a que todos hayan escrito el mensaje `Hello World`. Después, sólo uno de ellos escribirá el último mensaje.

1.1.3. Simple *tasking*

Una de las formas de paralelizar un programa es mediante la creación de tareas. Una *task* es un bloque de trabajo, es decir, una sección de código más el *data environment* en el que se creó que se ejecuta de forma asíncrona.

Por ejemplo:

```
int main(void) {
    int x, y, z;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        { x = f(); }       // la variable x recibe un valor
        #pragma omp task
        { y = g(); }       // la variable y recibe un valor
        #pragma omp task
        { z = h(); }       // la variable z recibe un valor
        printf("x = %d, y = %d, z = %d\n", x, y, z);
    }
}
```

Figura 3: Ejemplo de *tasking*

El *thread* que ejecute ese trozo de código creará tres *tasks* que serán en otro momento ejecutadas por otros *threads*. Si no añadimos ninguna directiva de sincronización únicamente podemos asegurar que `x`, `y`, `z` se habrán computado correctamente, pero no cuándo. En este ejemplo, puede ser que al llegar al `printf()` los valores aún no se hayan calculado.

1.1.4. Sincronización de *tasks*

Como una *task* se ejecuta de forma asíncrona, no podemos asegurar cuándo ha finalizado. Por tanto, necesitamos alguna directiva que nos permita esperar a su finalización.

1.1.4.1. *taskwait*

Esta directiva se encarga de hacer esperar al *thread* a que todas las *tasks* que ha creado (*child tasks*) hayan finalizado su ejecución. En el siguiente ejemplo muestra un ejemplo de su uso:

```
int main(void) {
    int x, y1, y2, z;
    #pragma omp parallel
    #pragma omp single
    {
        x = f();
        #pragma omp task
        { y1 = g1(x); }
        #pragma omp task
        { y2 = g2(x); }
        #pragma omp taskwait
        z = h(y1) + h(y2);
        printf("z = %d\n", z);
    }
}
```

Figura 4: Ejemplo de sincronización (*taskwait*)

Cuando se vaya a asignar valor a `z` ya tendremos los valores `y1` e `y2` calculados. Nótese que en caso de crear *tasks* dentro de otras (*descendant tasks*), *taskwait* no las esperará, ya que no han sido creadas por el *thread* que ha realizado el *taskwait*.

1.1.4.2. *taskgroup*

Esta directiva funciona igual que *taskwait* junto con el añadido de esperar también a las *descendant tasks*. En el siguiente ejemplo se muestra su uso:

```c
int main(void) {
    int x, y1, y2, z;
    #pragma omp parallel
    #pragma omp single
    {
        x = f();
        #pragma omp taskgroup
        {
            #pragma omp task
            {
                y1 = g1(x);
                #pragma omp task
                { y2 = g2(x); }
            }
        }
        z = h(y1) + h(y2);
        printf("z = %d\n", z);
    }
}
```

Figura 5: Ejemplo de sincronización (*taskgroup*)

En este caso, `y1` e `y2` tomarán valor antes de calcular el valor de `z`.

Si hubiéramos utilizado en este caso *taskwait*, únicamente podríamos asegurar que `y1` se ha calculado.

1.1.4.3. *Task dependencies*

Las directivas de sincronización anteriores pueden ser un tanto restrictivas en algunos casos, ya que los *threads* se ven obligados a bloquearse. Ese tiempo de espera podría dedicarse a otro tipo de trabajo. Desafortunadamente, el *runtime* no tiene la suficiente información para permitir que los *threads* continúen su ejecución garantizando que el resultado sea correcto.

El uso de dependencias nos permite mayor flexibilidad a la hora de definir el orden en el que las tareas deben completarse. Con la información suministrada al *runtime* con la cláusula `depend()`, se abren las puertas a otros escenarios de ejecución válidos. Por ejemplo, nos permite solapar creación de tareas con su ejecución, ya que el *runtime* será el que se encargue de ejecutarlas en el orden correcto.

La siguiente figura muestra un ejemplo de uso de esta cláusula:

```
int array[N];
int main(void) {
    #pragma omp parallel
    #pragma omp single
    {
        for (int i = 0; i < N; ++i) { // Tasks productoras
            #pragma omp task depend(out : array[i])
            array[i] = foo(i);
        }
        for (int i = 0; i < N; ++i) { // Tasks consumidoras
            #pragma omp task depend(inout: array[i])
            array[i] = bar(array[i]);
        }
    }
}
```

Figura 6: Ejemplo de dependencias

Usando dependencias podemos tener tanto *tasks* productoras como consumidoras. Esto no es posible con *taskwait*, ya que deberíamos poner la directiva entre los `for`, bloqueando la ejecución de las tareas consumidoras hasta la finalización de las tareas productoras.

1.1.5. *Task data - sharing attribute rules*

OpenMP dispone de reglas de visibilidad para variables usadas dentro de una directiva. Estas reglas se clasifican en predeterminadas, implícitas y explícitas. Una variable puede ser por ejemplo: *private*, privada para cada *thread* sin inicializar; *firstprivate*, privada pero inicializada al valor que tiene en el momento que se ejecuta la directiva; o *shared*, compartida entre *threads*.

El siguiente ejemplo muestra su uso:

```
int main(void) {
    int val = 0;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task firstprivate(val)
        { val++; }
        #pragma omp taskwait
        printf("%d\n", val);
    }
}
```

Figura 7: Ejemplo de *data sharing*

El valor del `printf()` será 0, ya que la *task* incrementa su propia variable `val`, sin modificar la original.

1.1.6. *taskloop*

Esta directiva se utiliza para paralelizar bucles dividiéndolos en *chunks*, o grupos de iteraciones, usando *tasks*. Este *construct* admite prácticamente las mismas cláusulas que la directiva *task*, como los *data - sharings*, `if()/final()`.

Además, *taskloop* permite especificar manualmente el tamaño de los *chunks* mediante las cláusulas `grainsize()`, que indica la cantidad de iteraciones que hará cada tarea; y `num_tasks()`, que indica la cantidad de tareas que se crearán para paralelizar el bucle.

A continuación se muestra un ejemplo:

```
int main(void) {
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp taskloop
        for (int i = 0; i < N; i++)
            do_something(i);
    }
}
```

Figura 8: Ejemplo de *taskloop*

2. Motivación

Desde el inicio de mis estudios de ingeniería, el uso constante de un compilador en gran parte de los proyectos académicos del grado me ha generado una inquietud por saber cómo funciona y se diseña. Durante el grado, he aprendido a utilizar *OpenMP* en la asignatura de Paralelismo (PAR), así como programar un pequeño *runtime* que implemente las llamadas a función que genera *GNU Compiler Collection (GCC)*[4] en Programación y Arquitecturas Paralelas (PAP). Para tener una foto global del proceso de compilación de un programa *OpenMP*, falta aprender cómo el compilador traduce las directivas a llamadas al *runtime*. El problema es que la curva de aprendizaje inicial para comprender su interior no es tarea fácil, ya que no es habitual encontrar documentación enfocada al principiante.

Gracias al Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC - CNS), que me ha ofrecido la oportunidad para formarme en este campo, voy a dedicar el proyecto a extender el compilador *Mercurium*[5] para poder traducir las directivas *OpenMP* al *runtime* de Intel/*LLVM*.

Mercurium es un compilador muy arraigado a labores de investigación. Es por eso que contribuir en un *software* de este tipo es interesante de cara a ofrecer otra herramienta que puedan utilizar tanto los desarrolladores de *Mercurium* como desarrolladores de aplicaciones científicas que pretendan utilizar este compilador.

También, dada la facilidad de extensión de este compilador en comparación con otros, ofrecer un mejor soporte a un *runtime* de renombre como el de Intel/LLVM puede facilitar el prototipado de nuevas mejoras en el estándar *OpenMP*.

3. Estado del arte

Dado que *OpenMP* es una especificación, existen múltiples implementaciones que la cumplan. Habitualmente, estas implementaciones están formadas por una librería y una parte del compilador. La librería es la encargada de gestionar el paralelismo en *runtime*, mientras que el compilador es el encargado de transformar las directivas del programa a llamadas de esta librería.

La *Aplication Programming Interface (API)* de cada librería suele ser distinta, y cada compilador sólo soporta la suya. Los compiladores más relevantes en cuanto a soporte de *OpenMP* son *GCC*, *Clang*[6] e *Intel Parallel Studio*[7].

3.1. *GCC*

El compilador de *GNU* realiza la traducción de las directivas *OpenMP* a la librería *libgomp*[8]. *GCC* 6.1 soporta *OpenMP* 4.5, por lo que en casi cualquier sistema *GNU/Linux* se puede sacar partido de la última versión del estándar.

La siguiente figura muestra un ejemplo de cómo traduce este compilador las directivas *OpenMP*.

```
#pragma omp for
for (size_t i = 0; i < N; i++) array[i] = omp_get_thread_num();
```

```
long start = 0, end = N, incr = 1, chunk = 0; long _start, _end;
if (GOMP_loop_static_start(start, end, incr, chunk, &_start, &_end)) {
        do {
                for (size_t i = _start; i < _end; i += incr)
                        array[i] = omp_get_thread_num();
        } while (GOMP_loop_static_next(&_start, &_end));
} GOMP_loop_end();
```

Figura 9: Código *OpenMP* transformado (*GCC*)

3.2. *Clang*

Clang es un *frontend* de C para *LLVM* que rivaliza habitualmente con *GCC* en cuanto a generación de código se refiere. Una de sus ventajas es que ofrece mejor diagnóstico en caso de que surja algún error, además de estar mejor diseñado que el compilador de *GNU*, cuya base de código es más complicada de entender.

En cuanto a *OpenMP*, *Clang* traduce las directivas a la librería *libomp* [9] que ofrece soporte para *OpenMP* 4.5. La siguiente figura muestra la traducción del mismo código que en la figura 9, pero utilizando este compilador.

```
enum sched_type sched = kmp_sch_static;
kmp_int32 liter = 0, lower = 0, upper = N - 1, stride = 1, incr = 1;
kmp_int32 chunk = 0;

__kmpc_for_static_init_4(&loc1, *global_tid, sched,
                         &liter, &lower, &upper,
                         &stride, incr, chunk);
for (size_t i = lower; i <= upper; i += incr) {
        array[i] = omp_get_thread_num();
}
__kmpc_for_static_fini(&loc1, *global_tid);
__kmpc_barrier(&loc2, *global_tid);
```

Figura 10: Código *OpenMP* transformado (*Clang*)

3.3. *Intel Parallel Studio*

El compilador de Intel se caracteriza por su eficaz generación de código, superando a *GCC* y *Clang*. Sin embargo, es un *software closed source*, por lo que no disponemos de su código fuente y necesitamos una licencia para utilizarlo.

Intel soporta *OpenMP* 4.5 traduciendo las directivas a la librería *libiomp5*. Esta librería es prácticamente la misma que la que utiliza *Clang*, por lo que comparten casi la misma *API*. Éste es el motivo por el que el código *OpenMP* que genera este compilador es parecido al que genera *Clang*.

3.4. *Mercurium*

La complejidad de los compiladores anteriores es muy grande para trabajar en entornos académicos y de investigación. *Mercurium* ha sido diseñado para abordar este campo ofreciendo la posibilidad de implementar nuevas ideas sin demasiado esfuerzo.

Este compilador proporciona una representación interna compartida de C, C++ y Fortran que puede extenderse mediante la adicción de nuevas fases de compilación implementadas como *plugins*. Las transformaciones de código pueden implementarse mediante la escritura directa de código fuente sin dejar de lado el método de construir los nodos a mano para casos más avanzados.

Mercurium se usa principalmente junto con el *runtime Nanos++* para implementar *OmpSs* y *OpenMP*, aunque también dispone soporte parcial para el *runtime* de Intel.

Un *runtime* maduro como el de Intel y la extensibilidad de *Mercurium* forman una combinación muy interesante para probar ideas nuevas y, en definitiva, contribuir de una forma u otra a la mejora de *OpenMP*.

4. Objetivo

El objetivo de este proyecto, como he mencionado, es extender *Mercurium* para soportar el *runtime* de Intel.

4.1. Estado actual

Actualmente, *Mercurium* ya dispone de soporte para algunas directivas *OpenMP*:

- *parallel*: ésta es la directiva que se usa para crear una región de código paralelo.
- *barrier*: esta directiva crea una barrera en la que los *threads* esperan a que el último llegue a ésta.
- *master*: permite definir una sección de código que sólo será ejecutada por el *thread master*, que tiene identificador 0.
- *single*: similar a la directiva *master*. El *thread* que llegue primero a la sección de código indicada como *single* será el que la ejecute.
- *for*: directiva utilizada para dividir las iteraciones de un bucle entre *threads*.

4.2. Abasto del proyecto

Debido a las dimensiones de este trabajo, creo conveniente añadir soporte únicamente a las directivas de *tasking*. Adicionalmente, se validará que el soporte actual de *Mercurium* funciona correctamente.

La pauta a seguir de cara a cumplir este objetivo es la siguiente:

1. Aprendizaje de la *API* interna de Intel/*LLVM*.
2. Aprendizaje sobre la estructura de *Mercurium*. Concretamente, la parte de *OpenMP* implementada.
3. Extensión de *Mercurium* soportando las directivas de *tasking*.

4. Elaboración de casos de uso/juegos de prueba que validen que tanto el punto anterior como el soporte previo al proyecto funcionan correctamente.

4.3. Obstáculos

Habiendo definido la trayectoria del proyecto debemos tener en cuenta qué obstáculos podemos encontrar y analizar su impacto en el mismo:

4.3.1. Falta de documentación

En el mundo del desarrollo de *software* es tarea imprescindible documentar, para que el usuario, o desarrolladores que quieran contribuir, puedan comprender exactamente cómo funciona. Lamentablemente es uno de los atributos que suelen faltar debido a múltiples causas, como la falta de tiempo.

Como consecuencia, los riesgos asociados a un mal uso del *software* (cuelgues en el sistema, resultados erróneos, etc.) aumentan. Por eso, es imprescindible asegurarse de cómo funciona el *software* que pretendemos utilizar.

En nuestro caso, la documentación del *runtime* de Intel asociada a la temática de nuestro proyecto no está redactada, por lo que en la planificación se establecerá una estrategia para mitigar este problema.

4.3.2. Errores de implementación y diseño

Los primeros diseños pueden dar lugar a implementaciones incorrectas. Es por eso importante escoger una metodología de trabajo que nos permita ir refinándolos cada poco tiempo.

Aunque disponer de un diseño estructurado y con sentido facilita mucho el trabajo, debemos tener en cuenta que pueden aparecer errores o ineficiencias en la implementación. De la misma forma que con el diseño, la metodología de trabajo deberá incluir, también, periodos de revisión. Aunque puede parecer que esto relentiza el desarrollo, nos evitará tener que invertir mucho tiempo en comprobaciones de funcionamiento.

4.4. Actores implicados

La influencia del proyecto afectará a varias personas:

4.4.1. *Main developer*

El responsable final es el desarrollador, que es el encargado del *management*, documentación y *testing* del proyecto. Todo esto debe tenerse en cuenta debido a su influencia en el presupuesto y el tiempo de desarrollo.

4.4.2. Directores

Eduard Ayguadé y Sergi Mateo son los responsables de supervisar y encaminar el proyecto validando que el plan de ejecución se cumpla.

4.4.3. Beneficiarios

- Desarrolladores de *Mercurium*, ya que éste puede ser uno de los primeros pasos de cara a poder contribuir al estándar de *OpenMP* con nuevas directivas. Utilizar un *runtime* que sea aceptado por las grandes empresas a la hora de presentar novedades puede facilitar su admisión.
- Usuarios que desarrollen aplicaciones científicas u otro tipo de software que utilice *Mercurium*.

5. Metodología

Antes de empezar el proyecto es interesante describir primero qué herramientas se utilizarán para ayudarnos a gestionar la carga de trabajo y lograr los objetivos en el periodo establecido.

5.1. Metodología de trabajo

En esencia, la pauta a seguir descrita anteriormente tiene tres puntos importantes: análisis, desarrollo y *testing*.

En el primer punto se analizarán en profundidad tanto el *runtime* como el compilador, que traducirá las directivas. Debido a la falta de documentación por parte del primero, se deberá de analizar el código generado por otros compiladores que lo soporten, como *Clang*, para entender el funcionamiento de cada llamada a la librería de *OpenMP*. Esto se complementará con un análisis del *source code*.

Habiendo completado el punto anterior, el segundo paso es realizar una primera implementación que posteriormente se irá refinando con la última etapa.

La metodología *eXtreme Programming*, de tipo *Agile* con iteraciones semanales se utilizará para garantizar que el proyecto avanza correctamente.

5.2. Herramientas de seguimiento

Git es la herramienta *de facto* que se utilizará para gestionar los cambios que se realizan en el código. Además, permite mantener copias del proyecto alojadas en servidores, minimizando la pérdida de información en caso de que ocurra.

Esto, junto a reuniones semanales con los directores evitará desviaciones del *scheduling*.

5.3. Método de validación

Como se ha descrito anteriormente, la elaboración de casos de uso que comprueben exhaustivamente, tanto lo que se espera como lo que no se espera del *software* desarrollado, es clave de cara a la corrección de errores. Por tanto, su realización irá a la vez que se va añadiendo soporte a la *API*.

6. Planificación

Tal y como indica la UPC, la cantidad de horas dedicadas a la realización de un trabajo de fin de grado equivalen a 15 *European Credit Transfer and Accumulation System (ECTS)*. Cada crédito equivale a unas 25 - 30 horas de trabajo, por lo que tenemos una suma de 375 a 450 horas.

Además, debemos añadir la carga de trabajo de *Gestió de Projectes (GEP)*, 3 créditos *ECTS*, con lo que tendremos un total de 450 - 540 horas.

A continuación se describe la planificación temporal.

6.1. Descripción de las tareas

En esta sección de especifican detalladamente las tareas a realizar.

6.1.1. Gestión de proyectos

La primera tarea es justo lo que estamos realizando en este momento, es decir, analizar y organizar el proyecto en sí. Ésta, engloba todas las entregas de *Gestió de Projectes (GEP)*, las cuales son:

1. Definición del abasto y contextualización del proyecto.
2. Planificación temporal.
3. Gestión económica y sostenibilidad.
4. Presentación preliminar.
5. Revisión de competencias.
6. Presentación oral y entrega del documento final.

6.1.2. Estándar *OpenMP*

Antes de nada, es importante conocer la especificación de *OpenMP* haciendo énfasis, sobretodo, en todas las directivas asociadas a *tasking*. Escribir pequeños juegos de prueba ayudará, además, a consolidar los conocimientos aprendidos.

6.1.3. Intel *API*

Una vez entendida la especificación de *OpenMP* debemos familiarizarnos con la *API* en la que trabajaremos. Por lo tanto, realizaremos una lectura de la documentación disponible con el objetivo de relacionar las directivas *OpenMP* y las funciones asociadas.

Es necesario remarcar que, debido a la falta de documentación de funciones de *tasking*, se deberá de examinar el código fuente.

6.1.4. *Mercurium* para usuarios

Instalar y aprender a usar el compilador junto con la librería de Intel es otra de las primeras tareas a realizar.

En esta tarea, se reescribirán los juegos de prueba que usan las directivas *OpenMP* utilizando la *API* de Intel directamente, comprobando que el resultado es el mismo.

Adicionalmente, como *Mercurium* puede generar código de Intel para algunas directivas *OpenMP*, podemos comparar sus resultados con nuestros programas. De esta forma podemos detectar errores en la implementación que después podremos solventar.

6.1.5. Interior de *Mercurium*

En esta tarea, se examinará el *source code* del compilador para entender su estructura. Debido a que este tipo de *software* es muy complejo, comprenderlo en su totalidad es muy difícil. Como nuestra labor se enfocará en la parte del *backend*, entender ésta será suficiente.

6.1.6. Entorno de desarrollo

Preparar el entorno de trabajo es importante de cara a maximizar la productividad. Por tanto, en esta parte se hará un *setup* de las herramientas necesarias, además de disponer ya de *Mercurium* y la librería de Intel.

En particular, se configurará *Git* (para el *tracking* de los avances) y el *build system*. Además, se configurarán las herramientas para poder usar MareNostrum[1]: *Secure SHell (SSH)*, *Virtual Private Network (VPN)*, etc.

6.1.7. Extender *Mercurium*

Esta tarea es la más importante. Debemos poner en práctica los conocimientos de las tareas anteriores para completarla con éxito.

Durante su curso, se harán modificaciones en el código del compilador según el plan establecido, comprobando y validando que los resultados sean correctos. Al ser un proceso iterativo, puede ser necesario volver a alguna de las tareas anteriores para poder seguir avanzando.

Se añadirá soporte a las directivas *OpenMP* en el orden siguiente:

1. Directiva *task*.
2. *Data - sharing attribute rules*

[1]MareNostrum es un *supercomputer* en el que probaremos alguna aplicación real para probar que nuestras transformaciones de las directivas *OpenMP* son correctas.

3. Directiva *taskwait*.
4. Directiva *taskgroup*.
5. Directiva *taskloop*.
6. Directiva *taskyield*.

Además, se corregirán los errores (si existen) encontrados en la sección 6.1.4.

6.1.8. Memoria final

Aunque la redacción de documento final se realiza durante todas las etapas, siempre se necesita un tiempo extra para unificar todas las partes del trabajo, reescribir texto para mayor coherencia y cohesión, etc.

Adicionalmente, se incluye la preparación de la defensa delante del tribunal.

6.2. Estimación de la duración de cada tarea

La siguiente tabla muestra para cada etapa su duración, el porcentaje de tiempo que representa respecto al total y el rol de la persona que la llevará a cabo. Como este proyecto dispone de un sólo desarrollador, éste deberá realizar todas las tareas.

Tarea	Duración (%)	Duración (h)	Rol
Gestión de proyectos	15	75	*Project Manager*
Estándar *OpenMP*	5	25	Programador
Intel *API*	5	25	Programador y *tester*
Mercurium para usuarios	4	20	Programador
Interior de *Mercurium*	16	80	Programador y tester
Entorno de desarrollo	2	10	Programador
Extender *Mercurium*	46	240	Programador y tester
Memoria final	7	40	*Project Manager*
Total	100	515	

Tabla 1: Distribución del tiempo y rol asociado

6.3. Dependencias entre tareas

El diagrama de Gantt del anexo describe las dependencias y su orden de ejecución.

6.4. Desviaciones de planificación

Aunque la elaboración de la planificación facilita el éxito del proyecto, es posible que se produzcan desviaciones. Teniendo esto en cuenta, debemos establecer unos planes de acción que se deberán seguir según la gravedad del desvío.

Es habitual que el desarrollador se encuentre con dificultades (como las mencionadas en la sección 4.3) que incrementen el tiempo invertido en alguna tarea. Un incremento de la carga de trabajo cuando el margen de desvío es pequeño (máximo dos semanas), puede solucionar el problema.

Por el contrario, si la gravedad de la desviación aumenta, ya sea por algún fallo en la planificación o por cualquier otro motivo, se discutirá con los directores qué decisión es más conveniente tomar para completar con éxito el proyecto.

6.5. Recursos

Durante el transcurso del proyecto, se utilizarán los siguientes recursos:

6.5.1. *Hardware*

- **Ordenador portátil:** Dell Latitude E7450 proporcionado por el BSC - CNS y un HP ZBook G2 personal.
- ***MareNostrum supercomputer.***
- **Cámara de vídeo:** o la del *smartphone* o la del portátil.

6.5.2. *Software*

- **Sistema operativo:** *Archlinux.*
- **Compilador:** *Mercurium*, *GCC*, *Clang.*

- ***Debugger:*** *GNU Project Debugger (GDB).*
- **Herramientas de desarrollo:** *Make* y *Autotools.*
- **Sistema de control de versiones:** *Git.*

6.5.3. Recursos humanos

A esta categoría pertenecen el desarrollador del proyecto y los directores que lo supervisan.

También, es importante tener en consideración a aquellos ingenieros que invierten su tiempo en aconsejar y guiar al desarrollador de este proyecto para lograr sus objetivos.

7. Gestión económica

En esta sección se describen las necesidades económicas del proyecto y se muestra el presupuesto final, necesario para conocer su viabilidad.

A continuación, se desgranan los costes (más impuestos) según su tipo, incluyendo su presupuesto.

7.1. Costes directos

7.1.1. Definición del coste

Según el Boletin Oficial del Estado (BOE)[10], la jornada de trabajo es de 1826 horas efectivas anuales. Teniendo esto en cuenta, las siguientes fórmulas se usarán para calcular las amortizaciones de los recursos usados en este proyecto:

$$coste/hora_{efectivo}\ (€/h) = \frac{\frac{Coste - Valor\ residual}{ud.} \cdot \#uds.}{Vida\ útil(años) \cdot 1826\ h}$$

$$coste\ imputado\ (€) = coste/hora_{efectivo} \cdot uso\ (h)$$

7.1.2. *Hardware*

Según la normativa de amortizaciones de equipos informáticos[11], el periodo de amortización máximo es de 8 años[12]. Sin embargo, debido a la naturaleza del *HPC*, reduciremos el intervalo a 4 años, ya que es un periodo razonable en el cual podemos declarar el *hardware* como obsoleto.

La siguiente tabla muestra los costes para cada elemento de esta categoría:

Recurso	ud./uds.	Coste ud. (€)	Valor residual (€)	Vida útil (años)	Uso (h)	Amortización (€)
Dell E7450	1	1469	300	4	140	22
HP Zbook	1	2500	350	4	400	117
Cámara	1	0[2]	-	-	1.5	0
MareNostrum	1	-[3]	-	-	10	-

Tabla 2: Costes *hardware*

7.1.3. *Software*

Los costes de adquisición y uso del *software* que necesitamos para llevar a cabo el proyecto son nulos. Esto es debido a que todos los recursos disponen de una licencia *Open Source*[4].

7.1.4. Recursos humanos

En el coste total debemos contar también a la persona que lo llevará a cabo, el desarrollador. En este caso, debido a que se dispone de un sólo trabajador que tomará diferentes roles durante el proyecto (gestor del proyecto, programador y *tester*), debemos escoger un sueldo (€/h) conveniente. La siguiente tabla muestra el sueldo y el tiempo dedicado a cada rol:

Rol	Salario (€/h)	Tiempo (h)	Coste (€)
Project manager	35	82	2870
Programador	26	323	8398
Tester	20	110	2200
Total	-	515	13468

Tabla 3: Desgranado de coste por rol y horas

[2]Si utilizamos la cámara del portátil no tenemos coste.

[3]No ha sido posible calcular el coste debido a falta de información sobre su consumo, coste de mantenimiento, vida útil y periodo de amortización.

[4]En resumen, permite que el *software* pueda ser libremente usado, modificado y compartido.

El sueldo por hora se ha recopilado, respectivamente, en las referencias [13], [14] y [15].

Por lo tanto, el sueldo medio del desarrollador es de $13468/515 \approx 26$ €/h

7.2. Costes indirectos

Adicionalmente a los costes explícitos, debemos tener en cuenta aquellos costes que afectan indirectamente al proyecto.

Concretamente, debemos tener en cuenta recursos como la electricidad, acceso a internet, etc. Sin embargo, como el lugar donde se desempeñará el trabajo es el BSC - CNS no es posible obtener dicha información, ya que es privada.

7.3. Imprevistos y contingencias

Aunque hemos planteado el proyecto sin dejar cabos sueltos, es posible que no hayamos tenido en cuenta alguna tarea que haga que nuestro presupuesto se dispare. Por eso, es importante disponer de un "fondo de reserva" para poder cubrir ese posible gasto adicional. Un 15 % de los costes directos/indirectos es razonable, teniendo en cuenta el tipo de proyecto.

Adicionalmente, existen riesgos relacionados con la planificación descrita en la sección 6.4. Como cualquier desviación de la planificación puede incrementar el presupuesto destinado a la tarea asociada, es necesario disponer de unos fondos extra. Nótese que esta desviación afecta a los recursos implicados en la tarea afectada. Será necesario, por tanto, recalcular las amortizaciones.

Como hemos comentado en la planificación, no esperamos grandes desviaciones de la ruta a seguir. Por tanto, creo que un 10 % de los costes directos/indirectos será suficiente.

7.4. Resumen

Sumando todo, el coste del proyecto se muestra en la siguiente tabla:

Concepto	Coste (€)
Hardware	139
Software	0
Recursos humanos	13468
Internet, electricidad...	-
	13607
Imprevistos	2041
Contingencias	1360
Total	17008

Tabla 4: Resumen de costes del proyecto

7.5. Control de gestión

Monitorizar los recursos utilizados en cada tarea así como el tiempo invertido en la misma es importante de cara a detectar desviaciones tanto de planificación, como de presupuesto.

Por eso, se realizarán reportes semanales que contabilizarán el tiempo invertido en cada tarea, así como el uso de los recursos en la misma. De esta forma, podremos detectar cualquier desviación en un periodo corto de tiempo.

8. Sostenibilidad y compromiso social

En esta sección evaluamos la sostenibilidad y el compromiso social desde el ámbito económico, social y ambiental. Esta evaluación consiste en responder, mediante una puntuación, una serie de preguntas relacionadas con varios aspectos de cada ámbito.

A continuación, se muestra la matriz de sostenibilidad como resultado:

	Proyecto en producción	**Vida útil**	**Riesgos**
Ambiental	Consumo del diseño	Huella ecológica	Riesgos ambientales
	6:10	18:20	0:0
Económico	Factura	Plan de viabilidad	Riesgos económicos
	7:10	15:20	0:0
Social	Impacto personal	Impacto social	Riesgos sociales
	8:10	17:20	0:0
Valoración total	21:30	50:60	0:0
		71:90	

Tabla 5: Matriz de sostenibilidad

8.1. Aspecto económico

La evaluación de costes, tanto materiales como humanos, está especificada detalladamente en la sección 7.

Además, se ha incluido en el presupuesto unos costes extras en el caso de producirse algún imprevisto o contingencia.

Los recursos utilizados en este proyecto son mínimos. Es por eso que es difícil poder plantear un presupuesto menor que cumpla con los requisitos necesarios para la realización de este trabajo.

Hay que tener en cuenta que este proyecto no está destinado al ámbito comercial, por lo que no tiene ningún carácter competitivo.

Adicionalmente, es necesario remarcar que este proyecto reutiliza y extiende el código de *Mercurium* asociado a la traducción de directivas *OpenMP* al *runtime* de Intel.

8.2. Aspecto social

Este proyecto tiene fines académicos a nivel personal. No sólo es el primer proyecto cuya magnitud supera a otros realizados como desarrollador, si no que también es el primero que es exigente en aspectos como la planificación y documentación.

Por otro lado, este proyecto pretende ser de utilidad en un futuro para contribuir al estándar de *OpenMP* y mejorar el rendimiento de aplicaciones paralelas que lo utilicen.

8.3. Aspecto ambiental

Todos los recursos utilizados en este proyecto son reutilizables, incluso después de su amortización.

Siendo un producto de tipo *software*, ningún material manufacturado será necesario, no se consumirá ningún recurso natural y no se producirá ningún tipo de contaminación.

Este proyecto permitirá aprovechar mejor los multiprocesadores, por lo que es posible reducir el consumo a través de la reducción del tiempo de ejecución.

MareNostrum es un *supercomputer* que sí tiene un impacto significativo en el medio ambiente. Por otro lado, nuestro proyecto simplemente usará en algún momento puntual un grupo pequeño de nodos durante poco tiempo. En definitiva, el impacto debido a las acciones de este trabajo es negligible.

En conclusión, nuestro proyecto: tiene una huella ecológica ínfima, puede llegar a reducir el consumo y nos permite realizarlo con unos recursos mínimos que cualquier usuario podría comprar[5].

[5]Salvo el uso de *MareNostrum*, que su uso se limita al personal del BSC - CNS y autorizados.

9. Visión general de un compilador

En este capítulo se explicarán algunos conceptos básicos de compiladores para luego presentar el pipeline de *Mercurium*.

Normalmente, nos referimos a un compilador como un programa que traduce de un lenguaje de alto nivel a código máquina. Aunque este tipo de traducción código fuente - código máquina es la más habitual, existen otros tipos, dando lugar a gran cantidad de tipos de compiladores. Por ejemplo:

- **Código fuente → código máquina (misma arquitectura):** El más habitual para ejecutar el programa en nuestra plataforma.
- **Código fuente → código máquina (otra arquitectura):** Habitualmente utilizado para programar *embedded systems*.
- **Código fuente → código fuente** Para traducir de un lenguaje de programación a otro, por ejemplo.

Mercurium es un compilador del tercer tipo, aunque no se encarga de traducir de un lenguaje a otro, si no de reescribir directivas *OpenMP* y *OmpSs* por llamadas directas a la librería que implementa ese modelo de programación.

Un compilador se estructura normalmente en tres módulos: el *frontend*, el *middle end* y el *backend*.

9.1. *Frontend*

Este módulo del compilador es el que se encarga de procesar el *input* y generar algún tipo de *Intermediate Representation (IR)*. En el *frontend* se distinguen tres partes bien diferenciadas: el *scanner*, el *parser*, y el analizador semántico.

9.1.1. *Scanner*

Esta parte del *frontend* se encarga del análisis léxico. Su tarea consiste en convertir las cadenas de texto en *tokens*.

El *scanner* puede generarse mediante uso de herramientas como *flex* y *lex*, utilizando expresiones regulares.

Esta etapa puede detectar errores de forma de símbolos, identificadores, constantes, etc.

9.1.2. *Parser*

Esta parte del *frontend* se encarga del análisis sintáctico. Su tarea es generar una estructura que represente el programa utilizando los *tokens* de la etapa anterior. Habitualmente esta estructura es un *Abstract Syntax Tree (AST)*.

El *parser* puede generarse mediante uso de herramientas como *bison* y *yacc*, utilizando *context-free grammars*, aunque los compiladores comerciales tienden a usar el suyo propio. El motivo es que este tipo de herramientas pueden ser limitantes cuando aspectos como la velocidad, el reporte de errores y la recuperación en caso de error son indispensables.

Esta etapa puede detectar errores de sintaxis, como el típico error de que falta un ';' en C.

9.1.3. *Análisis semántico*

Esta última etapa del *frontend* se encarga de comprobar que el programa escrito es legal haciendo comprobaciones que las etapas anteriores no han podido realizar, como por ejemplo:

- Las variables están inicializadas antes de ser referenciadas.
- Una operación tiene argumentos con un tipo permitido.

- Número correcto de argumentos en llamadas a función.

Para ello, se crea una tabla de símbolos que incorpora para cada identificador (símbolo) del programa información relacionada con su declaración, aparición en el código, etc.

Esta estructura se usa durante todo el proceso de compilación. Además, es la que se adjunta en los *object files* para aportar más información en las sesiones de *debugging*.

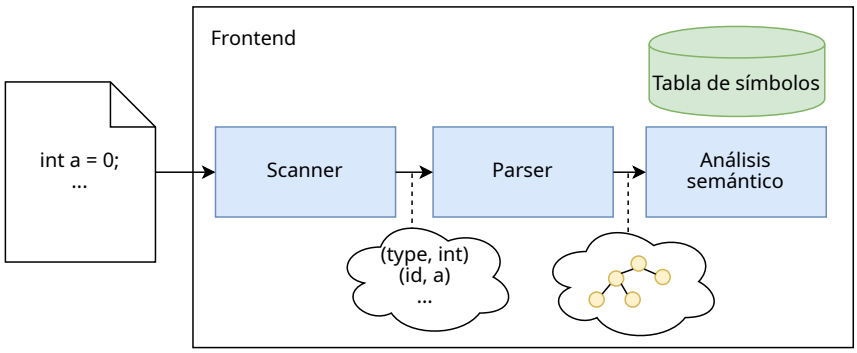


Figura 11: Representación gráfica del frontend

La salida de esta etapa suele ser una estructura de tipo árbol que incluye la información necesaria que represente el programa, además de información simbólica.

9.2. *Middle end*

Una vez ya se ha validado que el programa es correcto, el *middle end* se encarga de realizar optimizaciones independientes de la arquitectura, no sólo para generar código más eficiente, si no también para permitir al programador centrarse en la funcionalidad y portabilidad del mismo sin preocuparse demasiado por el tiempo de ejecución.

Algunas de las transformaciones que se realizan en esta etapa son:

- Descubrir y propagar constantes.
- Especializar código.
- Eliminar cálculos redundantes.
- Eliminar código no utilizado.
- Simplificar y combinar expresiones.

Normalmente, se puede indicar al compilador qué tipo de optimizaciones realizar dependiendo del objetivo que busquemos. Por ejemplo, normalmente los *embedded systems* tienen restricciones de memoria, por lo que quizá sea mejor reducir el tamaño del binario final sacrificando tiempo de ejecución.

9.3. *Backend*

Una vez ya se han realizado las optimizaciones independientes de la arquitectura pertinentes, el *backend* se encarga de generar el código objeto, y de realizar algunas optimizaciones dependientes del *target language*.

En primer lugar, se mapean las operaciones del *IR* a instrucciones del lenguaje máquina, simplificando expresiones complejas en otras de más simples. Por ejemplo, la operación `sin(x)` en X86 se puede realizar con una instrucción, mientras que en otra arquitectura es posible que se necesite emular por *software*.

En segundo lugar, se reordenan las instrucciones teniendo en cuenta el tiempo de ejecución de cada una, dependencias de datos, el pipeline del procesador, etc.

Por último, se posicionan las variables aprovechando la jerarquía de memoria (registro, cache, ram, disco...) de la arquitectura para la cual se genera el código. En este proceso se tienen en cuenta, por ejemplo, los registros que se deben guardar entre llamadas a función (MIPS), que algunos registros están formados por otros (X86), y por lo tanto no se pueden usar a la vez, etc.

10. *Pipeline* de *Mercurium*

En esta sección se presenta una visión general de cómo se estructura el compilador que utilizaremos en este proyecto.

10.1. *Path* básico

Cuando compilamos un programa secuencial el proceso de compilación es similar al de cualquier compilador. Concretamente, el comando que le indicamos a *Mercurium* llega al *driver*, el cual se encarga de desgranar el conjunto de caracteres identificando los parámetros para entender qué se quiere realizar.

Una vez el *driver* ha tratado toda la información, se encarga de orquestar el resto de componentes, como el *frontend*, el cual lee el fichero en cuestión, realiza los análisis léxico y sintáctico, generando un *AST*.

El análisis semántico recibe el *AST* y genera un árbol *Nodecl*. Esta nueva estructura organiza el contenido ligeramente diferente, ya que cierta información que existe en el *AST* no se materializa en el árbol *Nodecl*, si no que aparece dentro de lo que llamamos información simbólica.

Un ejemplo de árbol *Nodecl* es el de la figura 39 del anexo, generado a partir del código de la figura 38, también del anexo.

Por último, se ejecuta el *Codegen* el cual, utilizando la estructura mencionada anteriormente, genera el fichero de salida. Este fichero, si todo ha funcionamiento correctamente, será casi idéntico [6] al proporcionado en la entrada de *Mercurium*.

Al final, el *driver* ejecuta el compilador del lenguaje en el que esté escrito el fichero de entrada y se genera el binario.

[6] Es probable que se produzcan cambios de orden de declaración de variables u otro tipo de expresiones, aunque no deben afectar al funcionamiento del programa.

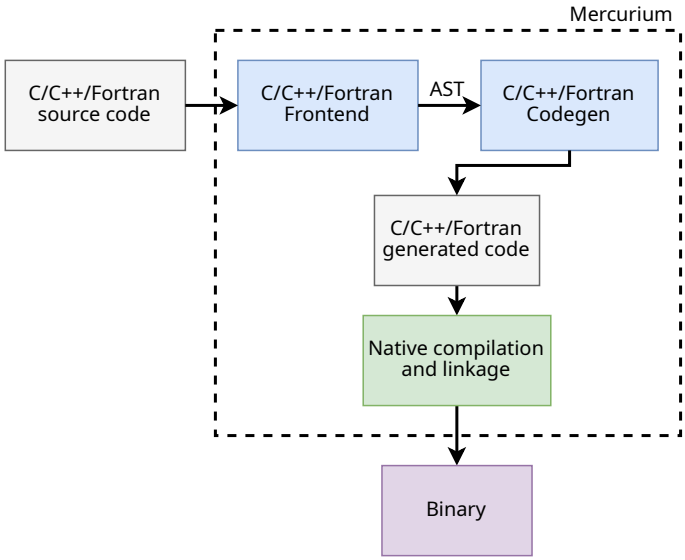


Figura 12: Representación gráfica del *path* básico

10.2. *Path* de *OpenMP*

Cuando compilamos un programa *OpenMP*, el proceso es el mismo hasta la generación del *Nodecl*. Esta estructura no tiene información sobre qué significan las directivas *OpenMP*. De hecho, el nodo correspondiente únicamente tiene un *string* con la directiva.

Si siguiéramos el mismo proceso que en el *path* básico, el *Codegen* daría como resultado el mismo código que el de la entrada, sin transformar las directivas.

La figura 39 del anexo de la sección anterior muestra en negro el nodo que contiene el *string* con la directiva *parallel*.

Continuando con la compilación, el *Nodecl* se pasa a un módulo del compilador que se encarga de analizar los nodos de las directivas y enriquecerlos con más información.

Por ejemplo:

1. Cuales son las variables compartidas y privadas.
2. El tipo de *scheduler* de un *for*.
3. Las expresiones a evaluar en un `omp task if(expresión) final(expresión)`.
4. El número de *threads* especificados explícitamente en un `omp parallel`.

Con toda esta nueva información ya tenemos todo lo necesario para transformar el código del programa para que funcione con la *runtime library* que nos interese. Ahora bien, si pasáramos este nuevo *Nodecl* al *Codegen* no obtendríamos el resultado que queremos. El motivo es que toda esta información no está especializada, es decir, no tiene un código asociado que lo represente. El *Codegen* únicamente sabe escribir *source code*, por lo que no sabe qué debería escribir en el caso de un nodo *OpenMP*.

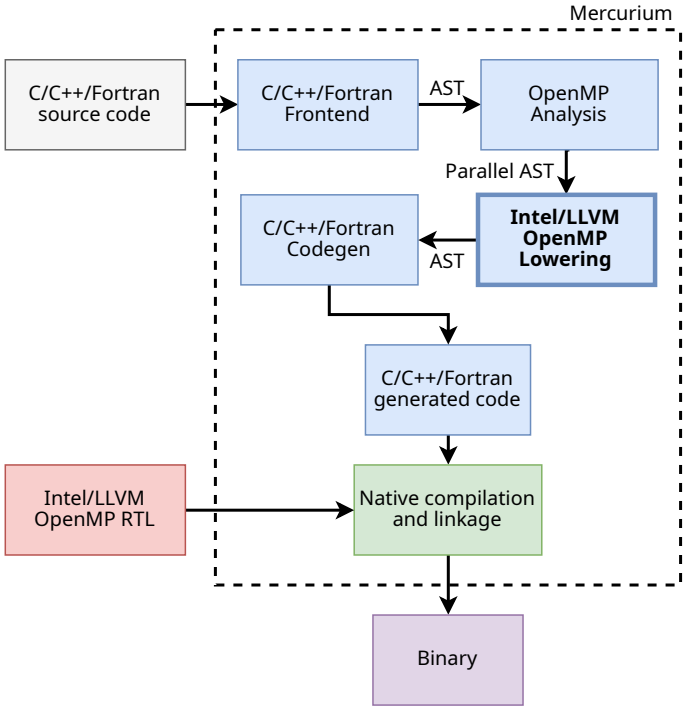


Figura 13: Representación gráfica del *path OpenMP*

La figura 40 del anexo muestra parcialmente el árbol generado para el mismo código anterior después de esta etapa de análisis. El nodo correspondiente al *parallel* está en negro. Nótese que de el cuelga el nodo de la lista de variables compartidas, si debe haber un *barrier* o no después de la directiva, etc.

La etapa de *lowering* es la que se encarga de substituir los nodos *OpenMP* por otros que, una vez generado el código final, interactúen con la librería que implemente *OpenMP* correspondiente y conseguir el funcionamiento paralelo que buscábamos.

Este proyecto se enfocará en la etapa de *lowering*, resaltada en la figura 13.

11. Intel/LLVM *OpenMP lowering*

Dado que prácticamente la totalidad del desarrollo de este proyecto se enfoca en la fase de *lowering*, es importante entender exactamente cuál es el objetivo de esta fase. Como se ha comentado brevemente en la sección anterior, esta fase se encarga de definir el comportamiento de los nodos *OpenMP*, los cuales no tienen ninguna representación a nivel de código. Concretamente transformaremos el código de manera que estos nodos desaparezcan, substituyéndolos por otros que realicen llamadas al *runtime*.

Mercurium permite realizar esto gracias al patrón del visitante. El compilador dispone de un visitante por defecto para cada nodo. Durante esta fase, *Mercurium* recorre exhaustivamente todo el árbol visitando todos los nodos, por lo que si implementamos nuestro propio visitante para el nodo *OpenMP* que nos interese, el compilador realizará las transformaciones que queremos.

11.1. *taskwait construct*

La mejor forma de entender cómo funciona la fase de *lowering* es viendo un ejemplo sencillo como *taskwait*.

Observando lo que dice la *API* de Intel vemos que únicamente necesitamos la siguiente llamada a función:

```
kmp_int32 __kmpc_omp_taskwait(ident_t *loc_ref, kmp_int32 gtid);
```

Figura 14: *taskwait* en Intel

Los parámetros que necesita la función son un `ident_t`, el cual es simplemente una estructura que tiene unos *flags* y una cadena de caracteres indicando el fichero, la línea y la columna donde está la directiva *OpenMP*; y un `global_tid` que podemos obtener mediante a una llamada a la siguiente función:

```
kmp_int32 __kmpc_global_thread_num(ident_t *loc);
```

Figura 15: Obtención del `global_tid`

Para realizar esta transformación debemos implementar un visitante en el cual construiremos los nodos *Nodecl* que representen la llamada a esa función, con sus argumentos.

Esto se puede lograr por dos vías: construyendo los nodos manualmente, o utilizando un `Source`. Un `Source` es una herramienta que pone *Mercurium* a nuestra disposición para automatizar el proceso de construcción de los nodos *Nodecl*. Nos permite escribir literalmente lo que queremos hacer tal y como lo haríamos al programar en un lenguaje de programación. Para obtener los nodos que representen lo que hemos descrito en el `Source`, éste se envía al *frontend*, indicándole el *scope* donde se situará para realizar los análisis léxico, sintáctico y semántico.

Por último, sólo nos queda reemplazar el nodo *taskwait construct* por el árbol que hemos generado ya sea manualmente o utilizando `Source` y pasándoselo al *frontend* de *Mercurium*.

11.2. *task construct*

Esta es la directiva más complicada, ya que aquí debemos gestionar el paso de las variables involucradas en la tarea según su tipo, si la tarea tiene dependencias, si tiene cláusulas `final(...)`, `if(...)`, etc.

Vamos a realizar una primera versión en la cual únicamente crearemos la tarea sin tener en cuenta *data sharings* ni otras cláusulas.

11.2.1. Primera iteración: creación de la tarea

Igual que con *taskwait*, veamos que funciones debemos utilizar para crear una tarea:

```
kmp_int32 __kmpc_omp_task(ident_t *loc_ref, kmp_int32 gtid, kmp_task_t *new_task);
kmp_task_t* __kmpc_omp_task_alloc(ident_t *loc_ref,
                                  kmp_int32 gtid,
                                  kmp_int32 flags,
                                  size_t sizeof_kmp_task_t,
                                  size_t sizeof_shareds,
                                  kmp_routine_entry_t task_entry);
```

Figura 16: *task* en Intel

Como podemos ver, debemos pedir memoria al *runtime* indicando la función donde está el código de la tarea en `task_entry`, el `global_tid` y el `ident_t`. Siguiendo la misma metodología que en *taskwait* substituimos la directiva por esas dos llamadas.

Lo único que nos faltaría sería crear la función de *task* y mover el código de la tarea a ésta. Este paso es trivial, ya que tenemos acceso tanto a los *statements* del cuerpo de la tarea, como los de la función que acabamos de crear.

En esta primera versión la transformación quedaría como sigue:

```c
int main(void) {
    #pragma omp parallel
    {   ...
        #pragma omp task
        { printf("Hello World!\n"); }
    }
}
```

Figura 17: Código a transformar, primera versión

```c
static void _task_main_0(kmp_int32 _global_tid, kmp_task_t *_task)
{ printf("Hello World!\n"); }

static void _ol_main_0(kmp_int32 *_global_tid, kmp_int32 *_bound_tid) {
    ...
    kmp_int32 cached_gtid_value_0 = __kmpc_global_thread_num(&_loc_1);
    {
        kmp_task_t *_ret;
        _ret = __kmpc_omp_task_alloc(&_loc_0, cached_gtid_value_0,
                                     1, sizeof(kmp_task_t), 0,
                                     (kmp_routine_entry_t)&_task_main_0);
        __kmpc_omp_task(&_loc_0, cached_gtid_value_0, (kmp_task_t *)_ret);
    }
}

int main(void)
{ __kmpc_fork_call(&_loc_2, 0, (kmpc_micro)_ol_main_0); }
```

Figura 18: Resultado del *lowering* de *task*, primera versión

11.2.2. Segunda iteración: *data - sharings*

Aunque la primera versión es funcional, no tenemos en cuenta las variables que aparecen fuera del *scope* de la tarea.

En esta segunda versión se soportarán las cláusulas *shared*, *private* y *firstprivate*.

Antes hemos visto que el *alloc* de la tarea nos devuelve un tipo `kmp_task_t *`. Dentro de él, entre otras cosas, hay un campo llamado *shareds* que podemos usar para pasar información a la tarea.

En el caso de las variables compartidas únicamente debemos pasar un puntero a cada una, mientras que en las *firstprivate* debemos hacer una copia a esta estructura.

Una vez rellenemos el `kmp_task_t *` con la información sólo nos queda recuperarla en el contexto de la tarea.

A continuación se muestra un ejemplo de uso de variables compartidas, privadas y *firstprivate*, y el resultado del *lowering*.

```
int main(void) {
    int a, b, c;
    #pragma omp parallel
    {   ...
        #pragma omp task shared(a) firstprivate(b) private(c)
        { /* Cuerpo de la tarea */ }
    }
}
```

Figura 19: Código a transformar, segunda versión

```
struct _args_task_main_0
{ int *a; int b; };

static void _task_main_0(kmp_int32 _global_tid, kmp_task_t *_task) {
    struct _args_task_main_0 *_args = (struct _args_task_main_0 *)(*_task).shareds;
    int *const _task_a = &(*(*_args).a);
    int *const _task_b = &((*_args).b);
    int _task_c;
    { /* Cuerpo de la tarea */ }
}

static void _ol_main_0(... , int *const a, int *const b) {
    ...
    _ret = __kmpc_omp_task_alloc(... , sizeof(kmp_task_t),
                                 sizeof(struct _args_task_main_0), ...);

    _args = (struct _args_task_main_0 *)(*_ret).shareds;
    (*_args).a = (int *) a;
    (*_args).b = (*b);

    __kmpc_omp_task(&_loc_0, cached_gtid_value_0, (kmp_task_t *)_ret);
}
```

Figura 20: Resultado del *lowering* de *task*, segunda versión

Como podemos observar, después del *alloc* de la tarea realizamos la captura de las variables involucradas.

Dado que ahora tenemos las variables de la tarea con nombres y tipos (*lvalue reference*) diferentes, en el cuerpo de la tarea debemos reemplazar los símbolos antiguos por los nuevos que hemos creado.

Esto podemos hacerlo rellenando un mapa de símbolos. *Mercurium* se encargará de hacer la substitución cuando hagamos la copia de los nodos a la función *task*.

El caso de las variables privadas es trivial, ya que sólo hay que añadir en la tarea la definición de ésta.

11.2.3. Tercera iteración: Variable Length Array (VLA)

Existe un tipo de variable en C que es un poco especial llamada *Variable Length Array (VLA)*. Este tipo de variable se caracteriza por tener un tamaño que se concreta en tiempo de ejecución. Un ejemplo de ello es un *array* cuyo tamaño es `argc`, `rand()`, etc.

Tal y como tenemos la segunda versión no contemplamos este caso. En este apartado se comentará la estrategia que usaremos para soportar este tipo de variable.

Lo primero que tenemos que tener en cuenta es qué información debemos pasar y de qué tipo. Tal y como ya estamos haciendo en la versión anterior, lo que debemos de pasar es un puntero al *VLA* en caso de que sea *shared*, o el *array* entero en caso que sea *firstprivate*. Además, debemos pasar información del tamaño del *array*. ¿Qué tipo de variable debería ser? Dado que el contenido de la variable del tamaño del *array* puede cambiar, debemos hacer una copia para evitar modificaciones.

Por lo tanto, la variable del tamaño del *array* será *firstprivate*.

La siguiente figura muestra visualmente el paso de información de la tarea:

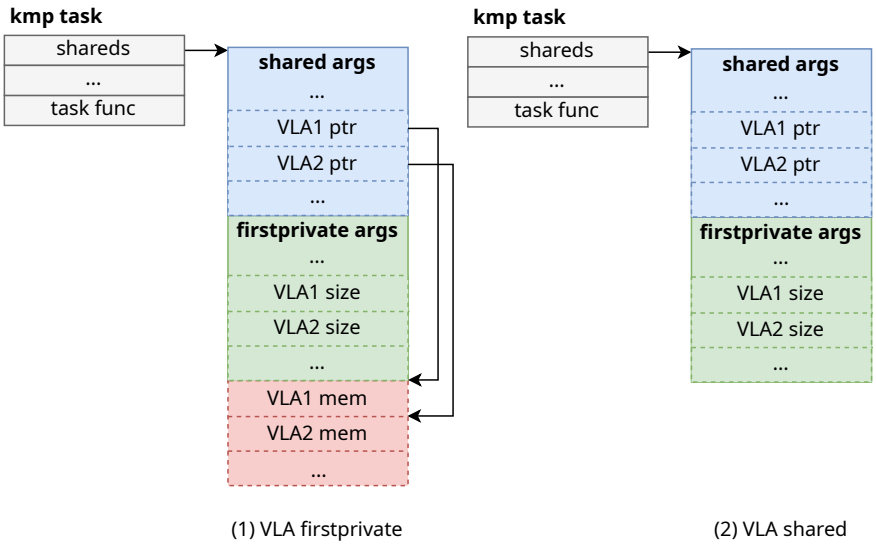


Figura 21: *Layout* de la tarea considerando *VLAs*

Cuando el *VLA* es compartido basta con tener el puntero a él y su tamaño, tal y como se ha comentado. La parte interesante está en el caso de que sea *firstprivate*. Como se puede observar, la estrategia consiste en hacer que ese puntero que pasaremos a la tarea apunte al final de la estructura donde tenemos una memoria extra. Esto funciona en *runtime* porqué cuando hacemos el *alloc* de la tarea ya conocemos el tamaño del *VLA*. En caso de tener varios, simplemente hay que tener en cuenta el tamaño que tienen los anteriores para evitar solapamientos.

Mercurium nos permite identificar si un *array* es un *VLA* o no, además de obtener las variables que contienen su tamaño. Para completar la versión, debemos añadir estas nuevas variables al mapa de símbolos.

Veamos el mismo ejemplo anterior únicamente con un *VLA firstprivate* llamado `array`, y el resultado del *lowering*:

```
struct  _args_task_main_0
{ long int mcc_vla_0; void *array; };

static void _task_main_0(kmp_int32 _global_tid, kmp_task_t *_task) {
    struct _args_task_main_0 *_args = (struct _args_task_main_0 *)(*_task).shareds;
    long int *const _task_mcc_vla_0 = &((*_args).mcc_vla_0);
    int *const _task_array = *((int (*)[(*_task_mcc_vla_0)])(*_args).array);
    { /* Cuerpo de la tarea */ }
}

static void _ol_main_0(... , const long int *const mcc_vla_0, int *const array) {
    ...
    _ret = __kmpc_omp_task_alloc(..., sizeof(kmp_task_t),
                                 sizeof(struct _args_task_main_0)
    /* Incremento de memoria */      + sizeof(int [p_mcc_vla_00]), ...);

    _args = (struct _args_task_main_0 *)(*_ret).shareds;
    (*_args).mcc_vla_0 = p_mcc_vla_00;
    (*_args).array = (char *)(_args + 1);
    __builtin_memcpy((*_args).array, array, sizeof(int [p_mcc_vla_00]));

    __kmpc_omp_task(&_loc_0, cached_gtid_value_0, (kmp_task_t *)_ret);
}
```

Figura 22: Resultado del *lowering* de *task*, tercera versión

Nótese que en el *alloc* pedimos la memoria extra necesaria para almacenar `array`, y luego hacemos que el puntero que utilizará la tarea apunte justo al final de la estructura.

11.2.4. Cuarta iteración: dependencias y cláusulas `if()` y `final()`

A la hora de soportar las cláusulas `if()`, `final()`, `untied`, etc., debemos darnos cuenta de cuándo se especifican y que expresión tienen (en el caso de `if()/final()`). El árbol *Nodecl* tiene unos nodos específicos cuando existen estas cláusulas en el código. Para obtenerlos debemos hacer un visitante de estos nodos y obtener la expresión que utilizaremos posteriormente.

En cuanto a las dependencias, el *runtime* espera un array donde cada posición contiene un puntero a la dependencia, su tipo (*in/out/inout*), y su tamaño en número de *bytes*.

A continuación se muestra un ejemplo que involucra una tarea con cláusula `if()` y dependencias:

```
int main(void) {
    int a; int x;
    #pragma omp parallel
    {
        #pragma omp task depend(out: a)
        { /* Cuerpo de la tarea */ }
        #pragma omp task if(x > 0) depend(in: a)
        { /* Cuerpo de la tarea */ }
    }
}
```

Figura 23: Código a transformar, primera versión

```
static void _ol_main_0(... , int *const a, int *const x) {
    {   ...
        _ret = __kmpc_omp_task_alloc(...);
        ...
        kmp_depend_info_t _deps[1L];
        _deps[0].base_addr = a; _deps[0].len = /* sizeof(int) */ 4;
        _deps[0].flags.in = 1; _deps[0].flags.out = 1;
        __kmpc_omp_task_with_deps(... , _ret,/* num_deps */ 1, _deps, ...);
    } { ...
        _ret = __kmpc_omp_task_alloc(...);
        ...
        kmp_depend_info_t _deps[1L];
        _deps[0].base_addr = a;
        _deps[0].len = /* sizeof(int) */ 4; _deps[0].flags.in = 1;
        if (*x > 0) {
            __kmpc_omp_task_with_deps(... , _ret,/* num_deps */ 1, _deps, ...);
        } else {
            __kmpc_omp_wait_deps(... , /* num_deps */ 1, _deps, ...);
            __kmpc_omp_task_begin_if0(... , _ret);
            _task_main_1(... , _ret);
            __kmpc_omp_task_complete_if0(... , _ret);
    }}}
```

Figura 24: Resultado de un `omp task depend() if()`

Cuando no se cumple la condición del `if()`, `__kmpc_omp_task_{begin, complete}_if0()` sirven para indicar al *runtime* que la tarea se ejecutará directamente.

Por otro lado, cuando hay dependencias se debe utilizar `__kmpc_omp_task_with_deps()` en lugar de `__kmpc_omp_task()`, y `__kmpc_omp_wait_deps()` cuando no se cumple la condición del `if()` para esperar a que se cumplan las dependencias.

11.3. *taskloop construct*

La directiva *taskloop* es una directa introducida en *OpenMP* 4.5 que facilita la paralelización de bucles mediante tareas. La ventaja de esta directiva es que permite especificar la repartición de las iteraciones fácilmente, en lugar de realizar la distribución a mano.

Inicialmente, esta directiva estaba soportada en *Mercurium* mediante el uso de las directivas *taskwait* y *task*, por lo que una vez implementadas teníamos soporte de *taskloop*.

Un ejemplo de transformación es el siguiente:

```
int main(void) {
        #pragma omp parallel
        #pragma omp single
        #pragma omp taskloop grainsize(5)
        for (int i = 0; i < 100; ++i)
                { /* Cuerpo del bucle */ }
```

Figura 25: Código a transformar, *taskloop*

```
static void _task_main_0(kmp_int32 _global_tid, kmp_task_t *_task) {
    struct _args_task_main_0 *_args = (struct _args_task_main_0 *)(*_task).shareds;
    int *const _task_omp_taskloop_0 = &((*_args).omp_taskloop_0);
    int *const _task_omp_block_0 = &((*_args).omp_block_0);
    for (int i = (*_task_omp_taskloop_0); i < (*_task_omp_block_0); i += 1)
        { /* Cuerpo del bucle taskloop */ }
}

static void _ol_main_0(...) {
    if (__kmpc_single(...)) {
        int omp_grainsize_0 = 5;
        for (int omp_taskloop_0 = 0;
             omp_taskloop_0 < 100;
             omp_taskloop_0 += omp_grainsize_0 * 1) {

            int omp_block_0 = omp_taskloop_0 + omp_grainsize_0 * 1;
            if (omp_block_0 > 100) omp_block_0 = 100;
            { /* Creación de la tarea */ }
        }
        __kmpc_omp_taskwait(...);
        __kmpc_end_single(...);
    }
    __kmpc_barrier(...);
}
```

Figura 26: Resultado del *lowering* de *taskloop*: primera versión

Como he mencionado, esta versión utilizaba *taskwait*, en lugar de *taskgroup*. Esto es debido a que esta implementación estaba diseñada para *OmpSs*. En *OpenMP* esta versión no era correcta debido a las diferencias entre esas dos directivas.

Para poder hacer el *lowering* de esta directiva correctamente, otros desarrolladores modificaron las fases del compilador previas a ésta. A continuación se comenta la implementación realizada.

Las funciones necesarias para la transformación son las siguientes:

```
kmp_task_t* __kmpc_omp_task_alloc(ident_t *loc_ref,
                                  kmp_int32 gtid,
                                  kmp_int32 flags,
                                  size_t sizeof_kmp_task_t,
                                  size_t sizeof_shareds,
                                  kmp_routine_entry_t task_entry);
void __kmpc_taskloop(ident_t *loc, kmp_int32 gtid, kmp_task_t *task,
                     kmp_int32 if_val, kmp_uint64 *lb, kmp_uint64 *ub,
                     kmp_int64 st, kmp_int32 nogroup, kmp_int32 sched,
                     kmp_uint64 grainsize, void *task_dup);
```

Figura 27: *taskloop* en Intel

Como se puede observar en la figura, el procedimiento es el mismo que con la directiva *task*. En este caso, además, debemos aportar más información: un *bool* para indicar si se realiza o no el *taskloop*, en caso de que se haya especificado la cláusula `if()`; el tipo de *scheduler* que se utilizará (cláusulas `grainsize()` y `num_tasks()`, etc.

El *runtime* espera que los *bounds* del bucle se pasen a la tarea como *firstprivate*. Además debemos indicarle su posición en la estructura mediante un puntero, de forma que al crear las tareas pueda modificar los *bounds* de cada una.

Lo único que falta es crear una función donde recuperaremos los *bounds* del bucle, que colocaremos a continuación.

```
struct  _kmp_taskloop__task_main_0
{ kmp_task_t task; int lb; int ub; int st; };
static void _task_main_0(..., struct _kmp_taskloop__task_main_0 *_taskloop) {
    int _task_i; int lower = (*_taskloop).lb;
    int upper = (*_taskloop).ub; int incr = (*_taskloop).st;
    for (_task_i = lower; _task_i <= upper; _task_i += incr)
    { /* Cuerpo del bucle */ }
}
static void _ol_main_0(...)
{   ...
    if (__kmpc_single(...)) {
        __kmpc_taskgroup(...);
        ...
        _ret = __kmpc_omp_task_alloc(...);
        (*_ret).lb = 0;  /* lower bound */
        (*_ret).ub = 99; /* upper bound */
        (*_ret).st = 1;  /* step */
        __kmpc_taskloop(... , (kmp_task_t *)_ret, 0,
                        &(*_ret).lb, &(*_ret).ub, (*_ret).st,
                        1, 1 /* scheduler: use grainsize */,
                        5 /* grainsize */, 0);
        __kmpc_end_taskgroup(...);
        __kmpc_end_single(...);
    } __kmpc_barrier(...);
}
```

Figura 28: Resultado del *lowering* de *taskloop*: segunda versión

Como se puede observar en la figura, como parámetros pasamos las direcciones de las variables que contienen el *upper bound* y el *lower bound*. De esta forma, el *runtime* sabe su posición relativa en la estructura y podrá especificar el *chunk* que le toca a cada tarea creada.

12. Control de calidad y evaluación de rendimiento

Una vez ya hemos completado el *lowering* de todas las directivas debemos comprobar y validar que todo funciona correctamente. Por otro lado, es interesante comparar la eficiencia de *Mercurium* transformando estas directivas con *Clang*, aprovechando que comparten la misma *API* en cuanto a *OpenMP runtime library* se refiere.

Para ello, se ha utilizado la *test suite* Bots[16], que dispone de varios *tests* que utilizan tareas.

12.1. Descripción del entorno

Para realizar tanto las pruebas de rendimiento como el *testing* se ha utilizado *MareNostrum IV*, el supercomputador más potente de España.

MareNostrum está formado por 48 *racks*, cada uno con 72 nodos de cómputo, o dicho de otra forma, con 3456 *cores* y 6912 *GB* de memoria. Estos nodos contienen dos *sockets* con Intel Xeon, 24 *cores* a 2.1 *Ghz* cada uno, y están interconectados mediante Intel Omni-Path.

Dado que vamos a correr *OpenMP*, utilizaremos un nodo de *MareNostrum* formado por dos *NUMA nodes*. Como no tenemos *HyperThreading* activado tenemos a nuestra disposición 48 *threads*.

Para compilar, se utilizarán *Clang 5.0* y *Mercurium* utilizando *Clang 5.0* para generar el binario. Ambos utilizarán el mismo *runtime* para comparar la eficiencia a nivel de generación de código.

12.2. Descripción de los *tests* y resultados

A continuación se da una breve introducción a los *tests* de Bots y se mostrarán los resultados.

12.2.1. *Alignment*

Este *test* se encarga de comparar secuencias de proteínas utilizando el algoritmo de *Myers - Miller*, que mediante programación dinámica y una estrategia de *divide and conquer* obtiene la secuencia óptima en $O(N)$.

Conceptualmente, tenemos una matriz en la cual la primera fila y la primera columna tendrán las secuencias a comparar. El método consiste en puntuar la coincidencia o no de cada par de aminoácidos considerando la puntuación de las subsecuencias anteriores, y penalizando los "espacios", o *gaps*, que se deben dejar para que coincidan.

El algoritmo calcula, para cada paso, el nodo de la columna del medio que pertenece al alineamiento óptimo mediante un *forward pass* en la mitad izquierda y un *reverse pass* en la mitad derecha. La siguiente muestra un ejemplo gráfico:

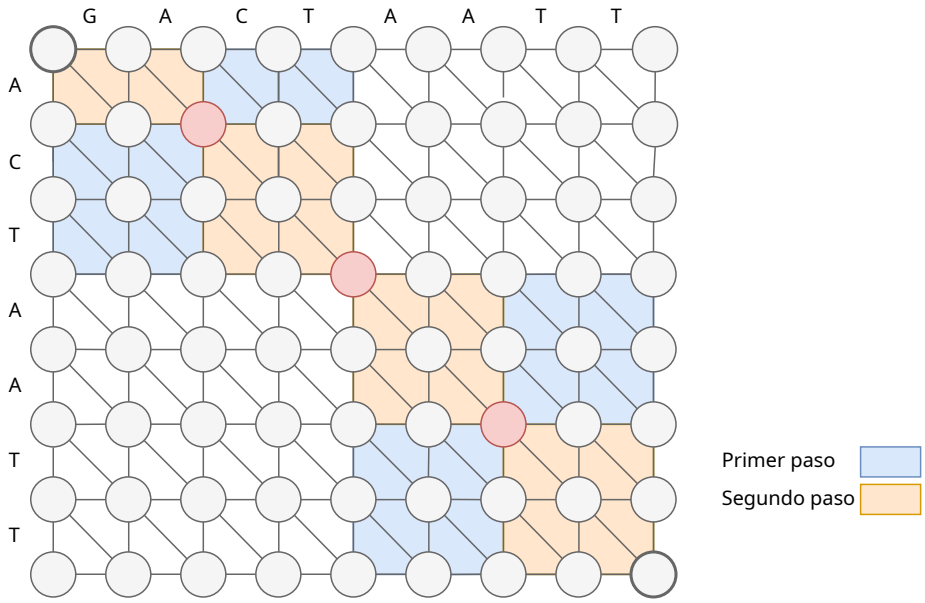


Figura 29: Ejemplo gráfico de alineamiento

Como se puede observar, una vez se ha encontrado el nodo del medio en la primera iteración se obtienen los de las dos mitades, y así hasta completar le secuencia.

La aplicación está paralelizada en el bucle que genera los pares de secuencias a comparar con un `omp for` y cada comparación se hace en una tarea.

La siguiente gráfica muestra el tiempo de ejecución obtenido con los binarios obtenidos de cada compilador:

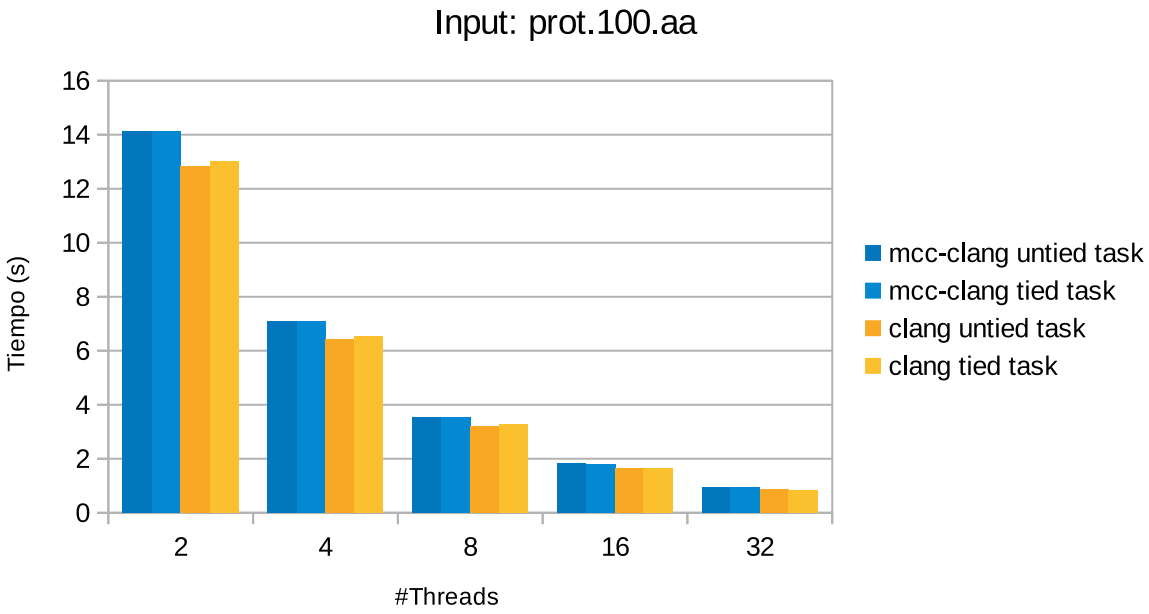


Figura 30: Resultados *Alignment*

Según los resultados, podemos ver cómo la implementación hecha en *Mercurium* está un poco por debajo de *Clang*, aunque no significativamente. Esto puede ser debido a que *Clang* genera el binario directamente, y es posible que decida hacer alguna pequeña optimización.

12.2.2. *Fast Fourier Transform (FFT)*

En este *test* se calcula la transformada de *Fourier* de una dimensión de un vector de n números complejos utilizando el algoritmo de *Cooley-Tukey*, que utiliza divide y vencerás para descomponer la *Discrete Fourier Transform*.

La aplicación está paralelizada generando tareas para cada paso recursivo, por lo que se debe esperar a que las tareas creadas en cada uno hayan finalizado antes.

La siguiente gráfica muestra el tiempo de ejecución obtenido con los binarios obtenidos de cada compilador:

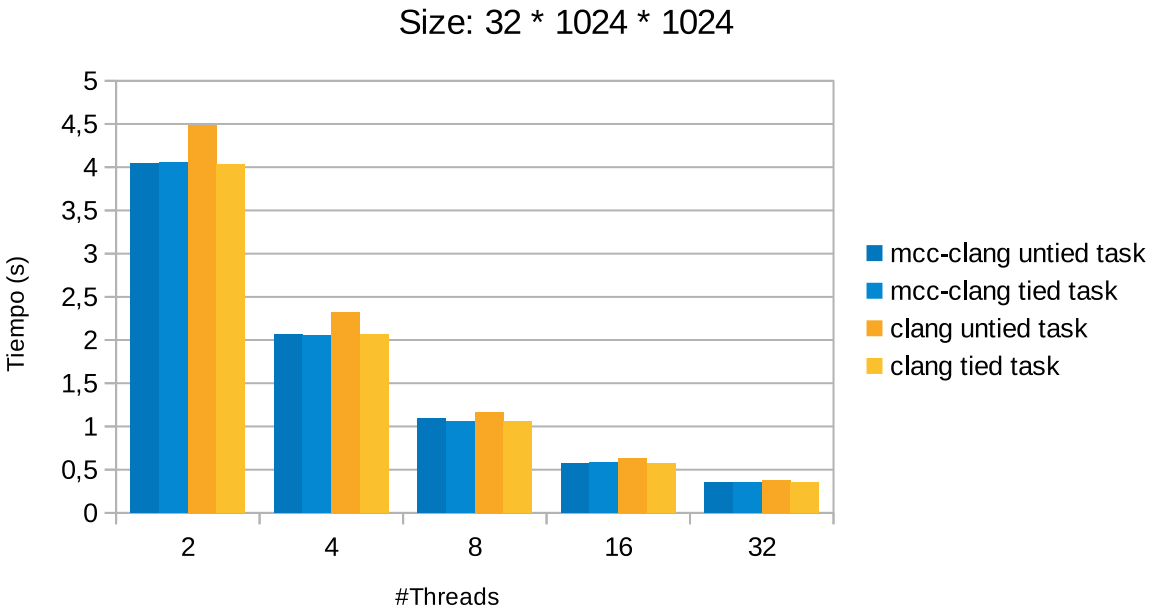


Figura 31: Resultados *FFT*

Igual que en la figura 30, los resultados son similares. En este caso parece que la versión de *Mercurium* con `task untied` es ligeramente más rápida que la de *Clang*.

12.2.3. *Health*

Este *test* simula el sistema sanitario de Colombia. Se utilizan listas multinivel donde cada elemento de la estructura representa un pueblo con una lista de pacientes potenciales y un hospital. El hospital tiene varias listas que representan el estado de cada paciente. A medida que avanza la simulación los pacientes enferman, reciben tratamiento, etc.

La aplicación está paralelizada utilizando tareas para cada pueblo. La siguiente gráfica muestra el tiempo de ejecución obtenido con los binarios obtenidos de cada compilador:

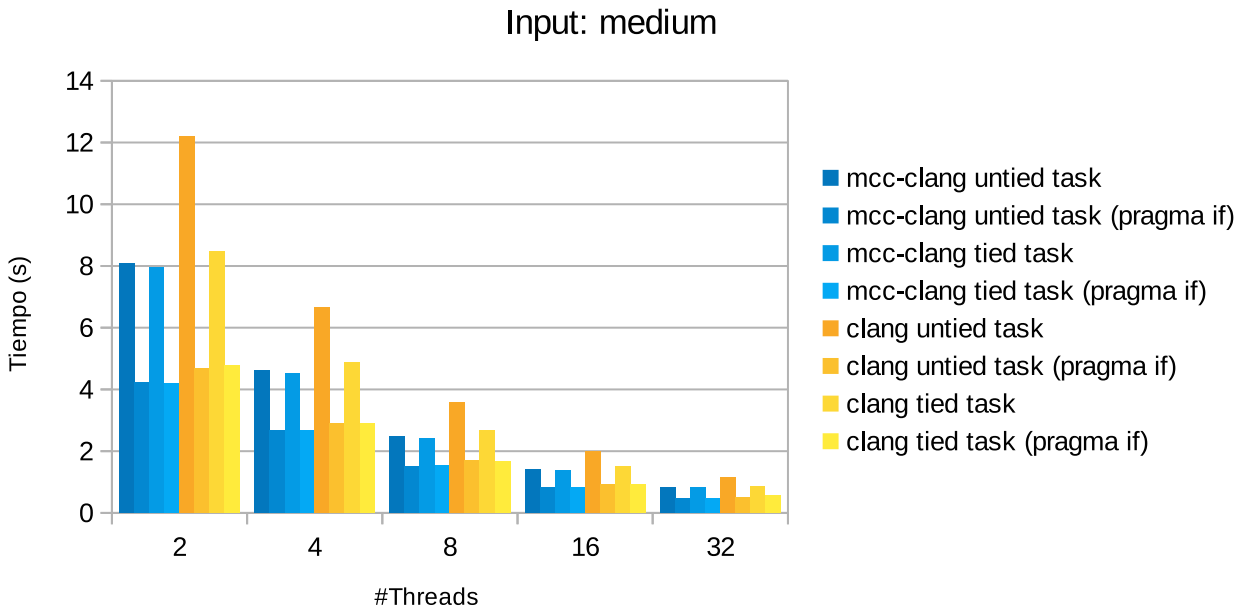


Figura 32: Resultados *Health*

En este *test* destaca la diferencia entre las versiones *mcc-clang untied task* y *clang untied task*, siendo mejor la primera. El resto de versiones no tienen diferencias significativas.

12.2.4. *Sort*

Este *test* ordena una permutación aleatoria de n elementos mediante un algoritmo de ordenación paralelo[17]. Primero, se divide el *array* de elementos en dos mitades y ordena cada una recursivamente. Segundo, se fusionan las dos mitades ordenadas mediante divide y vencerás.

El paralelismo se realiza creando tareas en cada división y en la fusión. Si el *array* es muy pequeño se realiza una ordenación secuencial.

La siguiente gráfica muestra el tiempo de ejecución obtenido con los binarios obtenidos de cada compilador:

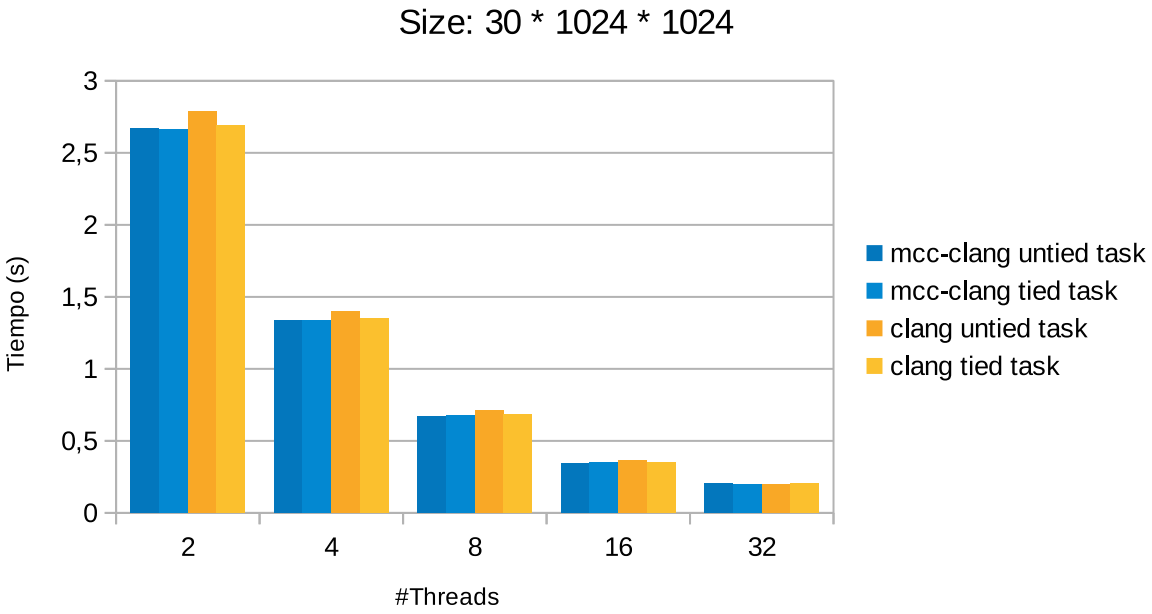


Figura 33: Resultados *Sort*

Como podemos ver, no hay diferencias significativas.

13. Conclusiones

El trabajo realizado durante este proyecto pretende contribuir a la comunidad involucrada en el desarrollo de *software* que utilice *OpenMP*, tanto a nivel de usuario como de desarrollador, extendiendo la fase de *lowering* de *Mercurium* a la *runtime library* de Intel/LLVM. El soporte de las directivas de *tasking*, objetivo de este proyecto, es uno de los primeros pasos de cara a completar y/o extender la especificación del estándar.

El primer paso realizado ha sido implementar la directiva `taskwait`, seguido de las directivas `task`, `taskgroup`, `taskyield` y `taskloop`.

Se ha completado con éxito el soporte de variables compartidas, privadas y *first-private*, destacando el soporte de *VLAs*. Además, se ha implementado el *lowering* de dependencias, así como las cláusulas `if/final/untied`.

Para corroborar la funcionalidad de la implementación, se han añadido *tests* a *Mercurium* para detectar errores en futuras modificaciones.

Por último, destacar que la ejecución con éxito de Bots y las pruebas de rendimiento realizadas en *MareNostrum* han demostrado que la generación de código está al nivel de otros compiladores de renombre.

14. Trabajo futuro

Aunque los objetivos definidos en este proyecto se han completado con éxito, aún quedan algunos aspectos del *lowering* de Intel en *Mercurium* por completar.

Una de las directivas todavía por implementar es *ordered*. Este *construct* es útil cuando al paralelizar un bucle queremos que ciertas partes de cada iteración se ejecuten tal y como lo haría un programa secuencial.

Otra directiva importante sin soportar actualmente es *atomic*, que nos permite utilizar las instrucciones específicas de la arquitectura para realizar accesos a memoria atómicos. Tal y como se comentó en una reunión con los directores, la transformación utilizará los *builtins* del compilador, aunque también existen funciones que hacen lo mismo en la *API* de Intel.

En cuanto al nuevo estándar *OpenMP* 5.0[18], todavía en desarrollo, existen nuevas características en el apartado de *tasking* que pueden soportarse. Una de ellas son las nuevas cláusulas *task reductions* y *in_reduction*, que permiten hacer reducciones tal y como se harían en un `omp for reduce()`, pero utilizando tareas.

Por otro lado, dado que el *runtime* de Intel parece desactualizado (última actualización del 2016), el primer paso de cara a soportar estas nuevas cláusulas sería substituir su librería por la de LLVM, que sí sigue en desarrollo. Ambas librerías son similares, por lo que no debe suponer un gran esfuerzo realizar este paso.

15. Revisión del proyecto

En la fase inicial de este proyecto se elaboró una planificación en la que se definieron unos objetivos que debían cumplirse para completar el trabajo con éxito. En esta sección se revisarán todos estos aspectos para verificar que todo ha salido como había sido previsto, y en caso contrario, indicar el motivo, si esta desviación ha afectado a otras áreas del proyecto, etc.

15.1. Planificación temporal

Inicialmente, el proyecto se dividió en pequeños objetivos que debían cumplirse junto con una estimación del tiempo necesario para completarlos.

La siguiente tabla muestra para cada objetivo el tiempo estimado y el dedicado:

Tarea	Tiempo estimado (h)	Tiempo dedicado (h)
Gestión de proyectos	75	75
Estándar *OpenMP*	25	10
Intel *API*	25	40
Mercurium para usuarios	20	10
Interior de *Mercurium*	80	80
Entorno de desarrollo	10	7
Extender *Mercurium*	240	250
Memoria final	40	40
Total	515	512

Tabla 6: Comparación del tiempo estimado y dedicado en las tareas del proyecto

15.1.1. Estándar *OpenMP*

Esta tarea de aprendizaje de la especificación *OpenMP* se completó antes de lo esperado. Esto se debe a que ya tenía conocimientos del modelo de programación, por lo que únicamente con hacer un par de juegos de prueba para refrescar la memoria fue suficiente.

15.1.2. Intel *API*

Esta tarea de aprendizaje de la *API* de Intel se completó invirtiendo más tiempo del previsto. La causa es que, aunque existe documentación, ésta es bastante pobre. No existen ejemplos de cómo debería usarse la *API* para traducir una directiva *OpenMP*.

Para paliar esto he tenido que compilar los juegos de prueba de la tarea anterior con *Clang*, que afortunadamente dispone de un *runtime* casi idéntico al de Intel. Una vez compilados he tenido que examinar el código ensamblador para entender la transformación que se debe realizar.

15.1.3. *Mercurium* para usuarios

Esta tarea únicamente consistía en instalar en el sistema el compilador y aprender a utilizarlo probando diferentes opciones en la línea de comandos. A la hora de compilar algún programa para probar el *lowering* de Intel, *Mercurium* no podía generar el binario final, ya que el perfil donde se indicaba qué compilador se debía usar estaba vacío. Al final no fue un problema grave y pude terminar la tarea antes de lo previsto.

15.1.4. Interior de *Mercurium*

Esta tarea de comprensión de la base de código se completó en el tiempo previsto, 80 horas. Aunque al principio tuve bastantes complicaciones, el codirector Sergi Mateo me solventó muchas de las dudas que tenía, además de darme una visión más global de cómo se estructura el compilador, las fases que tiene..., y me recomendó un documento que explicaba la estructura de un compilador genérico.

15.1.5. Entorno de desarrollo

La preparación de las herramientas necesarias para empezar a programar se completó antes de lo previsto debido a que se sobrestimó el tiempo necesario.

15.1.6. Entorno de desarrollo

Esta tarea es la que más tiempo ha tomado a realizarse. La directiva *task* es la más compleja que se ha tenido que implementar. Se realizaron varias reuniones con el codirector para decidir cómo abordar el soporte de las cláusulas *shared*, *private*, y *firstprivate*, así como determinar cómo debían tratarse los *VLAs*.

En cuanto a la directiva *taskloop*, surgió un imprevisto. Dado que *Mercurium* inicialmente transformaba esta directiva en fases previas al *lowering*, no era posible realizar la transformación específica de Intel. Mientras se realizaban los cambios pertinentes para permitir la transformación correctamente, se estuvo trabajando en la memoria.

Una vez se implementaron todas las transformaciones se utilizó Bots para validar su funcionamiento, así como realizar pruebas de rendimiento. Como tuve un par de problemas a la hora de ejecutar la *test suite* en *MareNostrum*, se invirtieron 10 horas más de lo esperado. Sin embargo, como iba bastante bien de tiempo no representó ningún problema.

15.2. Metodología

La metodología *eXtreme Programming*, de tipo *Agile* se escogió en la primera fase del proyecto.

Durante el transcurso del proyecto, reuniones periódicas con los directores se han realizado para mantener un ritmo de trabajo adecuado, además de permitir obtener *feedback* en cada paso realizado.

Como *Mercurium* es un proyecto en el que trabajan muchos desarrolladores al mismo tiempo, decidí trabajar en una rama paralela a la principal para que las modificaciones realizadas no tuvieran conflictos con el resto.

15.3. Leyes y regulaciones

Este proyecto en ningún momento obtendrá o almacenará datos y ficheros de carácter personal, y por lo tanto queda excluido de la ley orgánica 15/1999 de protección de datos de carácter personal (LOPD)[19].

En cuanto a las licencias *software*, *Mercurium* tiene una licencia *GNU Lesser General Public Licence 3.0*[20] que permite su uso comercial, así como su modificación y distribución.

16. Trabajo adicional

Cuando se estaban realizando las pruebas de rendimiento y el *testing*, me percaté que algunos fallaban debido a que la directiva *critical* no estaba implementada. En esta sección comentaré la transformación que he realizado para soportarla.

Observando la *API* de la *runtime library*, debemos utilizar las siguientes funciones:

```
void __kmpc_critical    (ident_t *, kmp_int32 global_tid, kmp_critical_name *);
void __kmpc_end_critical(ident_t *, kmp_int32 global_tid, kmp_critical_name *);
```

Figura 34: *critical* en Intel

Lo que debemos hacer es obtener el nodo *Nodecl* y mirar si se ha especificado un nombre para el *lock* que se le pasará al *runtime*. Para terminar, debemos colocar el *lock* en el *scope* global.

```
int main(void) {
    int a = 0;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp critical
        { a++; }
    }
}
```

Figura 35: Código a transformar, *critical*

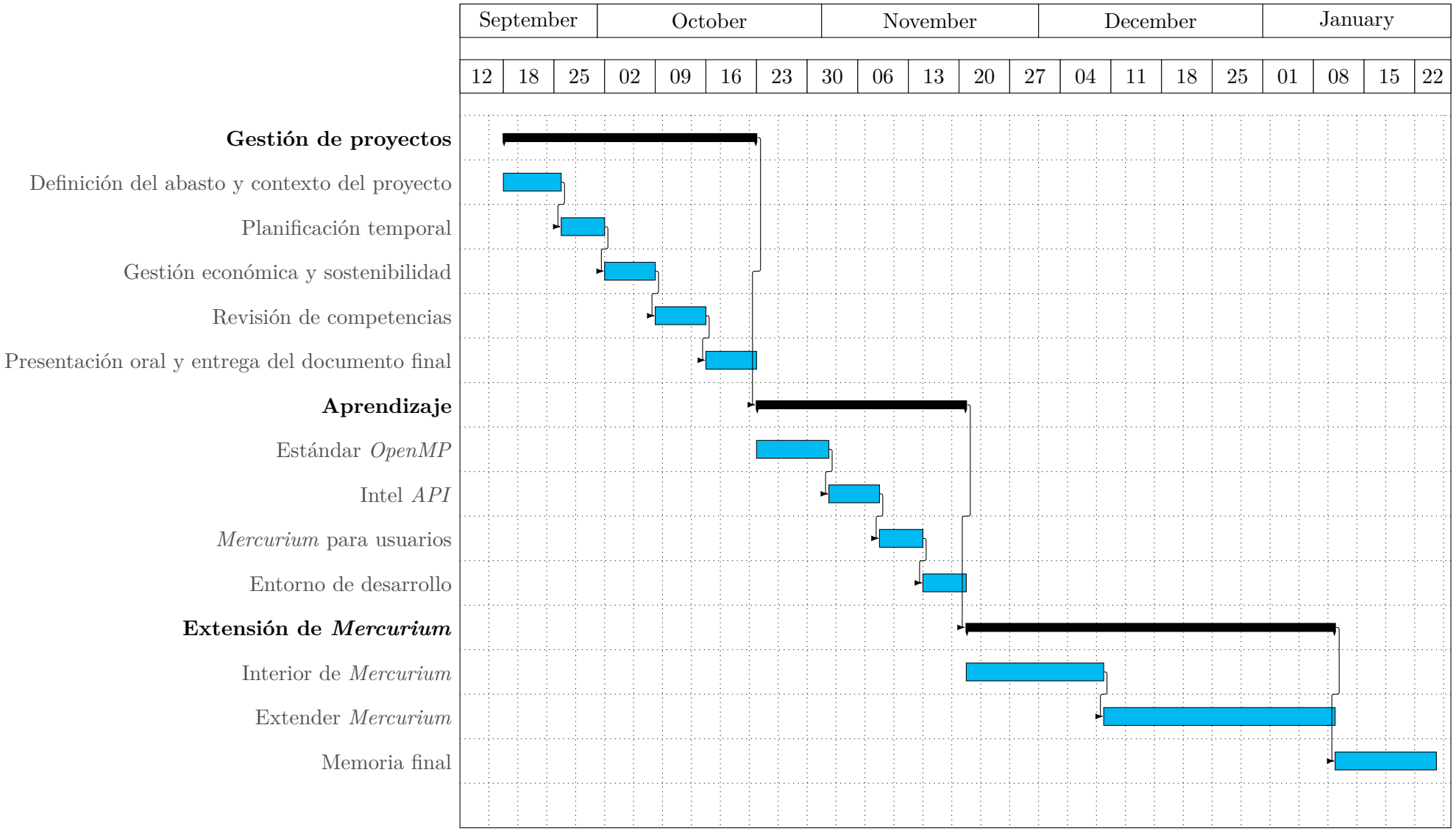
```
kmp_critical_name default_critical_lock;
static void _ol_main_0(... , int *const a) {
    __kmpc_critical(... , &default_critical_lock);
    { (*a)++; }
    __kmpc_end_critical(... , &default_critical_lock);
}
int main(void) {
    int a = 0;
    __kmpc_push_num_threads(... , 4);
    __kmpc_fork_call(..., 1, (kmpc_micro)_ol_main_0, &a);
}
```

Figura 36: Resultado del *lowering* de *critical*

Apéndice A Diagrama de Gantt

Tarea	Fecha inicio	Fecha fin	Cantidad de días
Gestión de proyectos	18-09-17	22-10-17	34
Definición del abasto y contexto del proyecto	18-09-17	25-09-17	8
Planificación temporal	26-09-17	01-10-17	6
Gestión económica y sostenibilidad	02-10-17	08-10-17	7
Revisión de competencias	09-10-17	15-10-17	7
Presentación oral y entrega del documento final	16-10-17	22-10-17	7
Aprendizaje	23-10-17	20-11-17	29
Estándar *OpenMP*	23-10-17	01-11-17	10
Intel *API*	02-11-17	08-11-17	7
Mercurium para usuarios	09-11-17	14-11-17	6
Entorno de desarrollo	15-11-17	20-11-17	6
Extensión de *Mercurium*	21-11-17	10-01-18	51
Interior de *Mercurium*	21-11-17	09-12-17	19
Extender *Mercurium*	10-12-17	10-01-18	32
Memoria final	11-01-18	24-01-18	14

Tabla 7: Descripción de inicio/fin de las tareas


September
October
November
December
January
12
18
25
02
09
16
23
30
06
13
20
27
04
11
18
25
01
08
15
22
Gestión de proyectos
Definición del abasto y contexto del proyecto
Planificación temporal
Gestión económica y sostenibilidad
Revisión de competencias
Presentación oral y entrega del documento final
Aprendizaje
Estándar OpenMP
Intel API
Mercurium para usuarios
Entorno de desarrollo
Extensión de Mercurium
Interior de Mercurium
Extender Mercurium
Memoria final

Apéndice B Topografía de *MareNostrum*

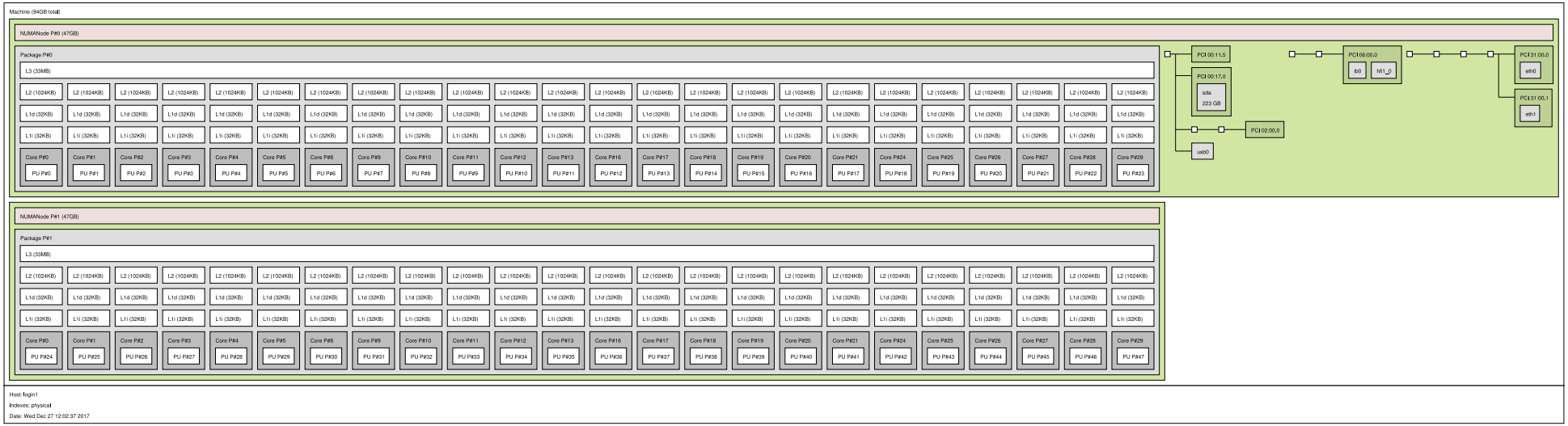


Figura 37: *MN4 lstopo output*

Apéndice C *Ejemplo árbol Nodecl*

```
int main(void) {
    int a;
    #pragma omp parallel
    { a = 0; }
}
```

Figura 38: Código sobre el que se obtienen los árboles *Nodecl*

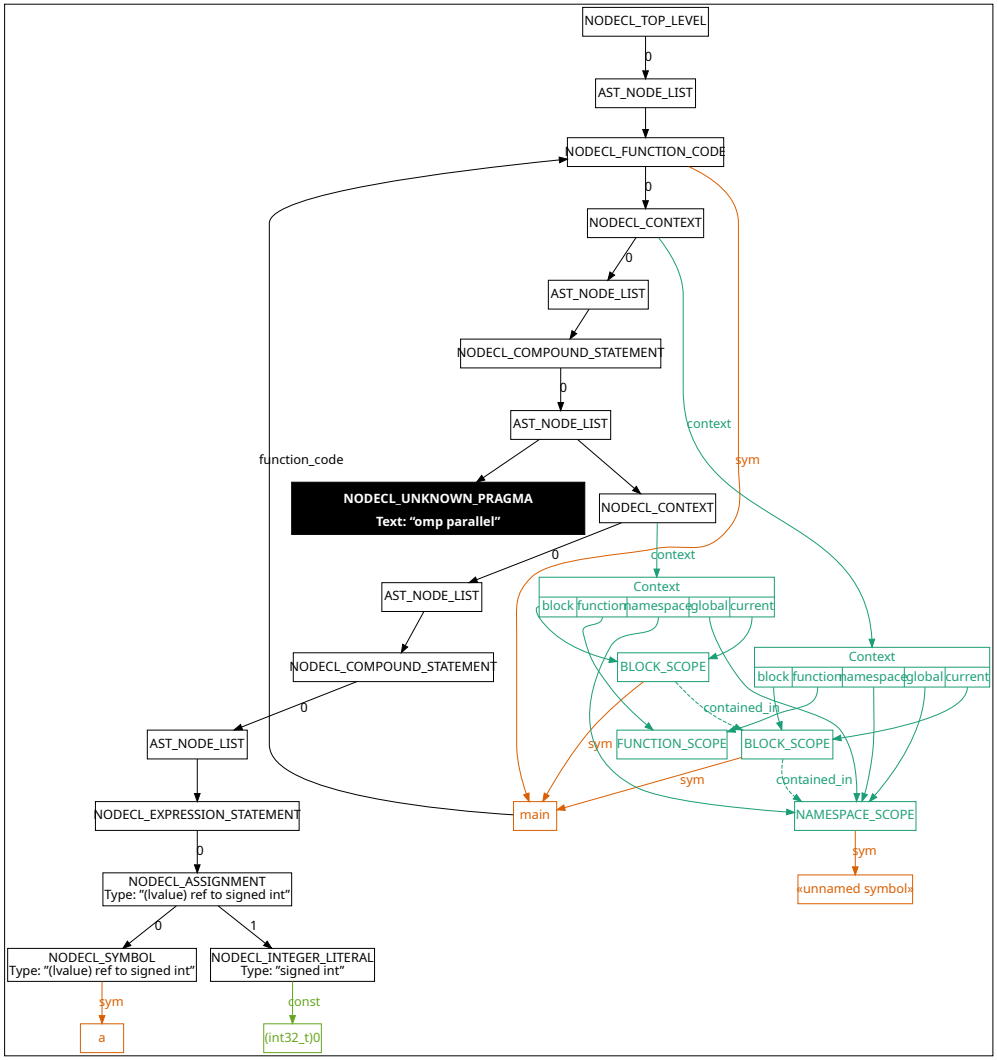


Figura 39: Árbol *Nodecl* sin *OpenMP*

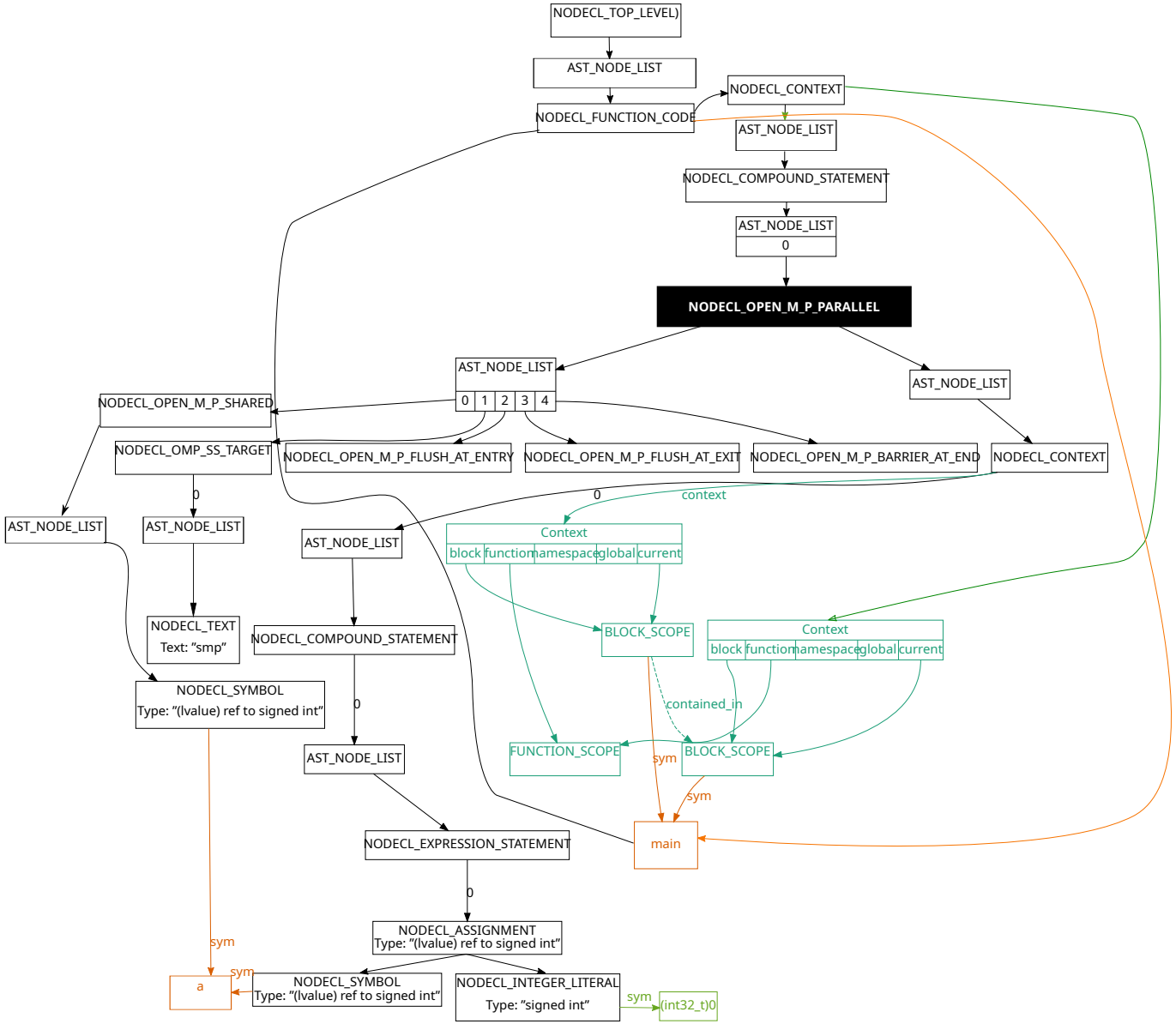


Figura 40: Árbol *Nodecl* sin *lowering*

Referencias

[1] High-performance computing (hpc) - european commission. https://ec.europa.eu/programmes/horizon2020/en/h2020-section/high-performance-computing-hpc. (Accedido el 09/25/2017).

[2] What is an fpga? field programmable gate array. https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. (Accedido el 09/25/2017).

[3] Message passing interface (mpi). https://computing.llnl.gov/tutorials/mpi/. (Accessed on 10/13/2017).

[4] Gcc, the gnu compiler collection - gnu project - free software foundation (fsf). https://gcc.gnu.org/. (Accedido el 09/25/2017).

[5] Mercurium | programming models @ bsc. https://pm.bsc.es/mcxx. (Accedido el 09/25/2017).

[6] Clang: C language family frontend for llvm. https://clang.llvm.org/. (Accedido el 09/25/2017).

[7] Home | intel® parallel studio xe | intel® software. https://software.intel.com/en-us/parallel-studio-xe. (Accedido el 09/25/2017).

[8] Gnu libgomp: Top. https://gcc.gnu.org/onlinedocs/libgomp/. (Accessed on 01/02/2018).

[9] Openmp* : Support for the openmp language. https://openmp.llvm.org/. (Accessed on 01/02/2018).

[10] Disposición 1989 del boe núm. 49 de 2015. https://www.boe.es/boe/dias/2015/02/26/pdfs/BOE-A-2015-1989.pdf. (Accedido el 10/09/2017).

[11] Sociedades 2016: manual práctico. http://www.agenciatributaria.es/static_files/AEAT/DIT/Contenidos_Publicos/CAT/AYUWEB/Biblioteca_Virtual/Manuales_practicos/Sociedades/Manual_Sociedades.pdf. (Accedido el 10/06/2017).

[12] Tabla de coeficientes de amortización lineal. - agencia tributaria. http://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2016/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml. (Accedido el 10/06/2017).

[13] Project manager, software development salary. https://www.payscale.com/research/US/Job=Project_Manager%2C_Software_Development/Salary. (Accessed on 10/23/2017).

[14] Software developer salary. https://www.payscale.com/research/US/Job=Software_Developer/Salary. (Accessed on 10/23/2017).

[15] Software tester salary. https://www.payscale.com/research/US/Job=Software_Tester/Salary. (Accessed on 10/23/2017).

[16] Github - bsc-pm/bots: Barcelona openmp task suite is a collection of applications that allow to test openmp tasking implementations and compare its behaviour under certain circumstances: task tiedness, throttle and cut-offs mechanisms, single/multiple task generators, etc. https://github.com/bsc-pm/bots. (Accessed on 12/20/2017).

[17] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, 36(11):1367–1369, 1987.

[18] www.openmp.org/wp-content/uploads/openmp-tr6.pdf. http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf. (Accessed on 01/11/2018).

[19] Boe.es - documento boe-a-1999-23750. https://www.boe.es/buscar/doc.php?id=BOE-A-1999-23750. (Accessed on 12/14/2017).

[20] Gnu lesser general public license v3 (lgpl-3.0) explained in plain english - tldrlegal. https://tldrlegal.com/license/gnu-lesser-general-public-license-v3-(lgpl-3). (Accessed on 12/14/2017).